jamk.fi

From Reflections to Systems Thinking

Developing a Practice for Documenting Software Architecture in a Growth Company

Jari Suni

Master's thesis
May 2020
School of Technology
Degree Programme in Full Stack Software Development

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Description

Author(s)	Type of publication	Date
Suni, Jari	Master's thesis	May 2020
		Language of publication: English
	Number of pages 93	Permission for web publication: x

Title of publication

From Reflections to Systems Thinking

Developing a Practice for Documenting Software Architecture in a Growth Company

Degree programme

Full Stack Software Development

Supervisor(s)

Rintamäki, Marko. Huotari, Jouni.

Assigned by

Woolman Oy

Abstract

It was identified by the assigner, Woolman company, that without architecture documentation the project and product development efforts struggled. Architecture documentation was business critical when proceeded from one phase to another: from sales to implementation and from projects to continuous services. The goal of the author was to develop a common practice for documenting software architecture in the company. The author studied architecture documentation in iterations and brought theory to practice through experimentation. The focus was to understand the essence of architecture work in the actual business context: when an architecture document should be produced, what should be documented and how.

The author used a diary-based method where he described his work and reflected his thinking, problem-solving, and learning. The relevant data for the qualitative research was gathered on a daily basis and followed up by a weekly reflective summary. The reporting period covered seven weeks in calendar time.

The results show that an all-encompassing architecture view is not worth pursuing. Instead, the problem needs to be systematically viewed from different perspectives using different architecture styles. To choose the appropriate set of views, one must identify the stakeholders that will depend on the documentation and understand each stakeholder's informational needs. Each architecture style has a different value depending on the viewer's context. Researching different architecture styles and applying them to practical work improved the author´s professional competence. Great development was made in the company´s documenting processes as well. These new operation modes described in the research can be used in any growth company or team.

Keywords/tags

software documentation, architecture style, domain-driven design, diary method

Miscellaneous

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Suni, Jari	Opinnäytetyö, ylempi AMK	toukokuu 2020
	Sivumäärä 93	Julkaisun kieli Englanti
		Verkkojulkaisulupa myönnetty: x

Työn nimi

Pohdinnoista systeemiajatteluun

Ohjelmistoarkkitehtuurin dokumentointikäytännön kehittäminen kasvuyrityksessä

Tutkinto-ohjelma

Full Stack Software Development

Työn ohjaaja(t)

Marko Rintamäki, Jouni Huotari

Toimeksiantaja(t)

Woolman Oy

Tiivistelmä

Toimeksiantaja, Woolman-yhtiö, havaitsi, että ilman arkkitehtuuridokumentaatiota projekti- ja tuotekehitys olivat vaikeuksissa. Arkkitehtuurin dokumentointi oli liiketoimintakriittistä erityisesti siirtymävaiheissa: myynnistä toteutukseen ja projekteista jatkuviin palveluihin. Tavoitteena oli kehittää yhteinen käytäntö ohjelmistoarkkitehtuurin dokumentoimiseksi yrityksessä. Arkkitehtuuridokumentaatiota tutkittiin iteraatioissa, ja teoriaa sovellettiin käytäntöön kokeilujen kautta. Tavoitteena oli ymmärtää arkkitehtuurityön ydin liiketoimintaympäristössä: milloin arkkitehtuuridokumentti tulisi tuottaa, mitä tulisi dokumentoida ja miten.

Tutkimuksessa käytettiin päiväkirjamenetelmää, jolla kuvattiin työtä, ajattelua, ongelmanratkaisua ja oppimista. Laadullisen tutkimuksen aineistoa kerättiin päivittäin, ja sitä seurasi viikoittainen analyysi. Raportointijakso oli seitsemän viikkoa kalenteriaikana.

Tulokset osoittavat, että kaiken kattavaa arkkitehtuurinäkymää ei kannata tavoitella. Sen sijaan ongelmaa on tarkasteltava systemaattisesti eri näkökulmista käyttämällä useita arkkitehtuurityylejä. Jotta voidaan valita sopiva joukko näkymiä, on ensin yksilöitävä sidosryhmät, jotka dokumentaatiota tarvitsevat, ja ymmärrettävä kunkin sidosryhmän tietotarpeet. Jokaisella arkkitehtuurityylillä on erilainen arvo riippuen katsojan kontekstista. Eri tyylien tutkiminen ja soveltaminen käytännön työhön paransivat tekijän ammattitaitoa. Myös yrityksen dokumentointiprosesseissa saavutettiin suurta kehitystä. Näitä tutkimuksessa kuvattuja uusia toimintamalleja voidaan soveltaa missä tahansa kasvuyrityksessä tai tiimissä.

Avainsanat

ohjelmistojen dokumentaatio, arkkitehtuurityyli, domain-driven design, päiväkirjamenetelmä

Muut tiedot

Contents

Figures

Tables

1 Introduction

1.1 Background and Motivation

The author works at Woolman-company which was founded in 2017. He joined the team in September 2017 as a tenth employee. Now the team has grown to 50 members (and still growing). Woolman has six locations, two of which are abroad. Indeed, the company has faced growing pains: procedures that worked well with 20 people did not work anymore with 40 people. The author has seen how their customer base has grown rapidly from dozens to hundreds. New team members are constantly introduced to the new and existing systems. Against this background it is probably easy to understand the main concerns related to their ecommerce project work: communication and resources, profitability and predictability, velocity and quality.

The company has identified that without architecture documentation the ecommerce projects and product development efforts stumble. A solid design is not enough. The design also needs to be documented and communicated. Architecture documentation is business critical when proceeded from one phase to another: from sales to implementation, and from projects to support. Every time a new team takes over, or a new member comes along, the challenges occur. As a company, they have to fix this to be able to grow.

Last year there was only one architect in the company. Now there is a team of three architects that reports to the CTO (Chief Technology Officer). The author also changed to a new role too and works now as a solution architect. The team have high expectations and a great amount to learn. The team need to find best practices for making design decisions and architecture documentation. However, documenting as such is not enough. Architecture must be communicated and understood by the stakeholders. How can this be achieved in an environment that is changing so rapidly?

Fortunately, software architecture is developed and researched diligently. Google Scholar finds 4 million hits for the search term “software architecture”. The author also made an online search on JAMK University of Applied Sciences’ library (Janet

database). International article search (PCI) returned almost 1.3 million results. Even the library's own collections showed over 200 results. Software architecture is a timely and constantly changing topic. However, since the assigner of the thesis is Woolman company, the author cannot spend years studying architecture books. He needs to set up an empirical process and try procedures out in the context of their company. How to bridge this gap?

1.2 Research Focus

The goal of the author is to develop a common practice for documenting software architecture in the company. The author studies the topic in iterations and brings theory to practice through experimentation. The focus is on understanding the essence of architecture work in the actual business context: when should an architecture document be produced, what should be documented and how?

According to Kananen (2015, 46), converting the problem into a research question makes work easier because it is easier to answer a question than a problem. Moreover, the format of research questions is important because a question provides solutions according to the question (ibid., 48). Kananen states (ibid.) that questions may be formulated as follows:

- What? (an explanatory question)
- How? (a question on a method)
- Why? (a question on a reason)

According to Kananen (ibid., 48), behind every question there is a fundamental question "what", and all other questions are sub-ordinate to that question. Without an answer to a what-question you cannot have other questions either. As an example, a how-question surveys depedencies or connections. In order to answer a how-question, you already need to know what it is about. (ibid., 48.)

Following the logic presented above, the research questions of this study are:

1. What are the best architecture styles and views for documenting ecommerce systems?
2. How should architecture document be produced effectively so that the team can learn from it and build a working ecommerce system from it?

These questions direct the author and the progress of research. As a result, the project work of the company, product development and especially team-to-team transitions should be easier. In general, less “friction” should be encountered than nowadays. The author should also improve his professional competence and become a better solution architect.

According to Kananen (2015, 51), data gathering methods for qualitative research are documents, observation and interviews. The primary sources of information for this thesis are:

- printed books and online articles about documenting software architectures
- interviews with specialists (e.g. developers and stakeholders) and communication with clients
- peer-to-peer feedback from co-architects
- experimenting in real life and reflecting the results.

1.3 Research Method

In this research the author uses a diary-based method where he describes his work and reflects his thinking, problem-solving, and learning. The relevant data for the research is gathered during February–April 2020 on a daily basis and followed up by a weekly reflective summary. The reporting period covers approximately 50 days in calendar time. It should also be noted that the following chapters (2-4) are written using the first person due to the diary method.

According to Bolger, Davis, and Rafaeli (2003, 580), diaries are self-report instruments used repeatedly to examine ongoing experiences in everyday situations. A fundamental benefit of the diary method is that it offers the opportunity to investigate events, experiences and processes in their natural context (ibid., 580). For the author of this thesis the context is, of course, his working life and the challenges that arise from it.

According to Bolger et al. (ibid., 581), three types of research goals can be achieved using the chosen method:

- obtain reliable person-level information
- obtain estimates of within-person change over time
- conduct a causal analysis of those changes

For the author of this thesis the desired change is an increase in professional skills and competence. In fact, the chosen research method provides a good basis for evaluating changes because the diary data are longitudinal. As Bolger et al. state (2003, 602) even if the investigator has no direct interest in time as a factor, the data are ordered in time and this ordering may be relevant to analyses. Diary entries in chapter 3 strictly follow this logic.

According to Sheble and Wildemuth (2009), some diaries incorporate non-textual accessories such as photographs and other digital objects. In other words, a diary might be a collage of text and non-text. In this thesis the non-textual elements such as diagrams are particularly important, and the reader should pay close attention to them. For this very reason, the diagrams have not been taken out of the context and moved, for example, to appendices.

What makes the diary method special in the context of this research is that the author analyzes the entries he has written himself. In other words, the author makes use of the material he has produced during the reporting period to carry out the actual analysis in the last chapter (Discussion). In this sense, it can be said that the thesis has features of autoethnographic research. The author especially likes the definition of the method by Ellis, Adams and Bochner (2011) as both a process and a product: *"a researcher uses tenets of autobiography and ethnography to do and write autoethnography"*.

According to Doloriert and Sambrook (2012, 83), autoethnography is derived from Greek, and it essentially means to write (research) about a nation (group of people) and the self (the researcher). The authors consider that the "ethnographic gaze" is a valuable tool for understanding organizational settings, processes and demands placed upon employees. Furthermore, they state that autoethnography enriches this research perspective by holding the researcher and organizational culture together. This is the practical reason why the diary method was chosen for this qualitative research. (Doloriert & Sambrook 2012, 83-86.)

On the other hand, the chosen method brings some ethical considerations for writing and reviewing the diary thesis. For example, the author may protect his colleagues, the organization, and its clients, and thus distort reporting. Secondly, it is not always

clear whether the others have understood the dual role of the author as both a participant and a researcher. This can put some pressure on what results to be published. (Doloriert & Sambrook 2012, 88.)

According to Chan (2016), the quality of autoethnography should be assessed according to the following principles:

1. Does it use authentic and trustworthy data?
2. Does it follow a reliable research process and show it clearly?
3. Does it follow ethical steps to protect the rights of self and others presented in the autoethnography?
4. Does it analyze the broader meaning of the personal experiences?
5. Does it attempt to make a scholarly contribution with its conclusion?

The author has tried to apply these principles in his work and bring out how the diary-based method creates understanding, through what process this understanding is created, and how it is applied to the reality itself.

2 Current State Description

2.1 Current State of Work

My job is to design and deliver ecommerce solutions for our clients. I support different functions of the company such as sales, projects and continuous services and the work assignments vary accordingly. In the pre-sales phase my main task is to support sales and find the best solutions to customer needs. However, any proposed solution must be balanced to fit the potential customer case. In other words, if the sales lead is a multinational corporation then the resources and with them the possibilities are likely to be vast. On the other hand, mature business processes and on-premise systems impose certain conditions that cannot be ignored.

If the sales lead is a solo entrepreneur or a startup, then the initial situation is very different. Existing systems do not limit solutions; however, investment capacity and resources can be very scarce. Whatever is the architecture of the solution, it must be justified to the customer and communicated to the rest of the team as well. This is especially important if the lead becomes a deal, and the proposed solution will be

implemented in practice. In fact, documentation of the sales phase has been an Achilles heel for us.

In addition to making proposals to potential customers, I also participate in the implementation work on a daily basis. I think this is a good way to maintain my ecommerce development expertise. As an architect, I both gather and provide information for the project team. I have an active role in this phase to ensure a comprehensive understanding of the project. I discuss the pros and cons of the options and try to make sure that the solution satisfies the business concept. I could describe my approach as risk-based: if there is a dark corner somewhere, I will try to illuminate it at an early stage. Whatever challenges we find and whatever we decide to do to them, documenting the decisions is important. In this matter we still have lessons to learn.

Last but not least, I work closely with our support team. Problems occur from side to side: one customer may have a problem with Google Shopping Ads, the other might need help with tax reports. The support ticket may be related to the development of a mobile application or the establishment of a loyalty program. Or maybe it is just confusion in printing shipping labels in the back room of the warehouse. One way or another, customer needs usually combine into one thing, urgency. The positive thing is that usually we find a solution quickly. The negative thing is that documenting the solution is often forgotten. As a result, the knowledge does not necessarily become the shared property of the company.

2.2 Stakeholders and Communication at Workplace

My main internal stakeholders are the sales team, project teams and support team. I work with them on a daily basis. However, as a solution architect, my work is very customer oriented in nature. I consider our clients to be my most essential stakeholders. Typically, I go through the specification of requirements and solution suggestions with an e-commerce manager or similar. The third major stakeholders are the technology partners of Shopify ecosystem. As an example, Shopify App Store includes thousands of applications that can be added to the online store. They serve as a good source of solutions to merchants' business problems. Each application has its strengths and weaknesses that need to be assessed at the system level. One could

say that reading documentation, technical comparison, numerous demos and practical experiments are the daily life of a solution architect.

Interaction with my colleagues varies quite a lot. Sometimes situations are determined by urgency, sometimes by dedication. For example, the team may receive a support ticket that must be taken care of immediately. In contrast, a big sales case can take a few years, including several definition phases. In the project implementation phase, the communication within the team is usually intensive and short-cycled. In any case, similarities can also be found. As an architect, I would like to see each of these interactions as fundamentally customer-driven and solution-focused. This is simply because we usually have a business problem to solve. I consider this analytical approach to be the best value I can bring to these situations. Of course, it is not easy, and it does not always work. When there is too much unknown, and the cause and effect are unclear, it is impossible to give a right answer. Then one just needs to act systematically to establish order in chaos, so to speak. In these situations, clear communication and documentation of decisions are further emphasized.

3 Diary Reporting

This chapter is the body of the thesis. The diary entries are arranged for weekly periods. Each week ends with an analysis that reflects on the past week and possibly anticipates the next. The best documenting practices are studied and implemented iteratively via the Plan-Do-Check-Adjust cycle. As an example, software architecture styles and notations identified in professional literature are applied to the business problems at hand. The concrete results like architecture views are then evaluated by the diarist and adjusted according to these observations.

3.1 Week 1

3.1.1 Monday 17 February 2020: Economics of Documentation

Today we have an internal weekly meeting. Team members attend from all locations. We have just started to develop new processes regarding the architecture work, and I am preparing a presentation about the topic for the meeting.

I start with *why* – why we are making this effort and try to get better at documenting architecture work. We have created core company values earlier, and I see a ponderable link between the topics. Our company values are

- Fun and passion
- Freedom, responsibility and power
- Care
- Learn and educate
- Plain language

I choose three of these values and consider the values from the solution architect's point of view. I am also referring to "audiences for architecture documentation" introduced in the book *"Documenting Software Architectures – Views and Beyond"* by Paul Clements and his colleagues (2010, 12-14).

- *Learn and educate.* Good architecture documentation serves as a means of education. It introduces the people to the system.
- *Plain language.* Architecture documentation is a vehicle for communication. It should be valuable both for the development team and the client. From the architect's point of view the documentation serves as a depository of thought.
- *Freedom, responsibility and power.* Third use case for an architecture documentation is to tell the team what to implement. It is the basis for system analysis and construction.

Our development/project teams are making decisions independently, so it is crucial to be aware of the consequences of those decisions over time and space. Easy way out usually leads back in, so to speak.

Another interesting question is *when* – when should you produce an architecture documentation? Every time you make any kind of a design decision? Probably not. Clements and others write about the economics of documentation in their book (Clements et al. 2010, 18). The formula they come up with is simple, even naïve:

(Cost of X without AD - Cost of X with AD) > Cost of AD

In this case the "cost of AD" is the time and money spent on producing and maintaining a particular architecture documentation. "Cost of X" is the cost of performing an activity, e.g. developing a new feature or fixing a defect, without and with the document. In other words, the payback should exceed the effort.

What does this really mean? It is impossible to predict all the development activities that take place in future. Nevertheless, I think that this is the essence of “economics of architecture documentation”. As an architect, you can never directly experience the consequences of many of your decisions. You need to be aware of this and think beyond the presence. In short, try to allocate your resources rightly, listen to the feedback, learn from mistakes and improve over time.

I also understand that there are at least two more questions that should be answered. Firstly, what is the desired outcome of the architectural work. Secondly, how the work is done *de facto.* These topics must be covered later on.

3.1.2 Tuesday 18 February 2020: Support Process

Today we have a weekly meeting with the architect team – a new ritual I have set up. We need to make our work visible and we are using Kanban boards for this. One board is for internal architecture work and it is called “Research”. Another board is for customer-oriented work. It is called “Sales support”. Here is an example, how the research board looks like (Figure 1):

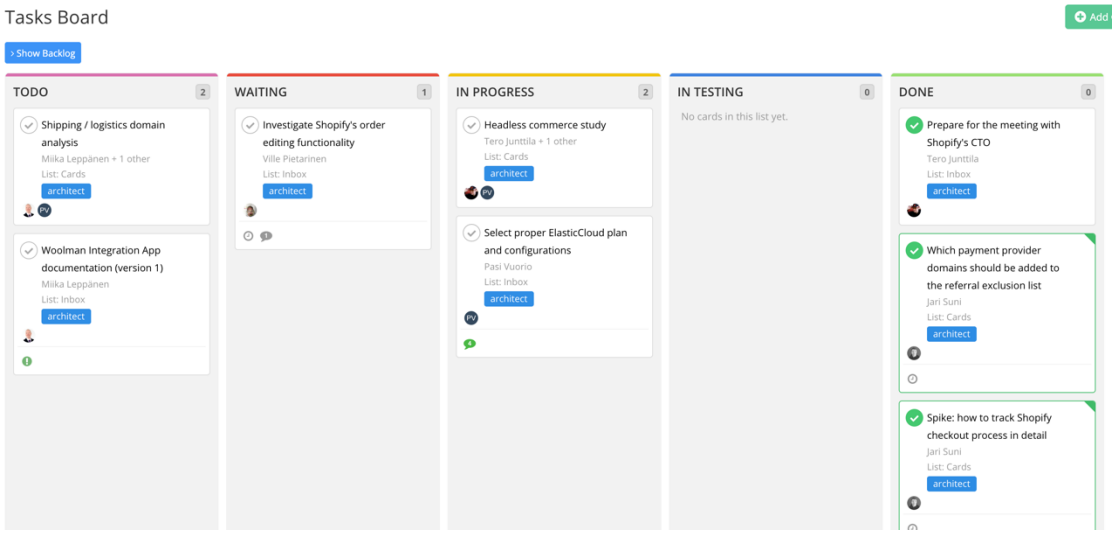


Figure 1. Kanban board

The squad for the meeting is

- CTO

- Full stack architect
- Solution architect (that's me)
- Shopify integration architect

My objectives to the meeting are

- refine enough backlog items so the architect team can be productive in the next sprint (week)
- establish shared vision and understanding about what is really important
- identify early major goals, and plan also longer term if possible

The weekly meeting is time-boxed for 30 minutes. Within that time, we go through the boards backwards – from DONE column to TODO. For some reason this feels more logical as we can first check what is done, then what is in progress, and finally what should be done next. As a result, we have an up-to-date prioritized task list for the upcoming week.

The process is far from perfect, but at least we have a process what to improve. As an example, we probably need someone from the sales team to prioritize the customer-cases for us. Now this is done mainly by "gut feeling".

3.1.3 Wednesday 19 February 2020: Tools for Documentation

I have been thinking that what would be a good document repository for our architecture work. It should be fun to use, easy to maintain, accessible and also look nice. Today I had a great discussion with our service manager who is responsible for our support services. He has been thinking the same issue from the support team's perspective. He came up with the initial list, and we made some additions to it from the architect's point of view. Here is the full list of the tools that we are going to use for documenting.

- **Teamwork Spaces**: Planning, specification and sharing knowledge both internally and externally. Main platform for architecture documentation.
- **Teamwork Projects**: Project management, project planning and task breakdown. Invoicing and time tracking.
- **Teamwork Desk**: Service management as support tickets. Public documentation for our SaaS products such as Nordic Shipping App.
- **Lucidchart**: Virtual workspace and our main tool for architecture design. As an example, I use it for data flow diagramming and system visualization. These documents will be embedded to Teamwork Spaces whenever needed.

- **HubSpot**: Our CRM and master for customer data. Customer management, customer contact information, customer communication.
- **GitLab**: Version control and deployment pipeline. Technical documentation and development guidelines that are close to the code.
- **Proposify**: Sales pipeline for managing leads, creating proposals and closing deals.
- **Google Drive**: Files storage. Coworking on files in real-time.

The big picture looks something like this (Figure 2):

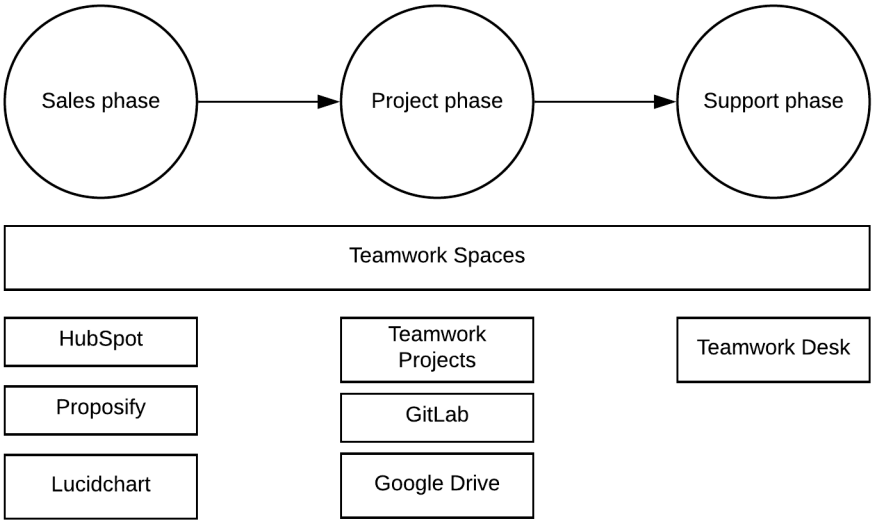


Figure 2. Documentation tools

In this sense Teamwork Spaces is an umbrella that tells us what the client wanted and how we understood it. It gathers the fragments together in a useful form. It's going to be our own Wikipedia. We agreed that if you ever need to check Slack message history, you probably should have put it in Spaces.

3.1.4 Thursday 20 February 2020: Levels of Architecture Domain

An exciting day ahead. We have second ever Woolman Architects Day. I hope we make it our monthly ritual. We are discussing what are the essential classes or levels of architecture domain for our work. We are brainstorming together and recognize three different levels. We write down the themes that characterize each level. As a result, we get a picture like this (Figure 3).

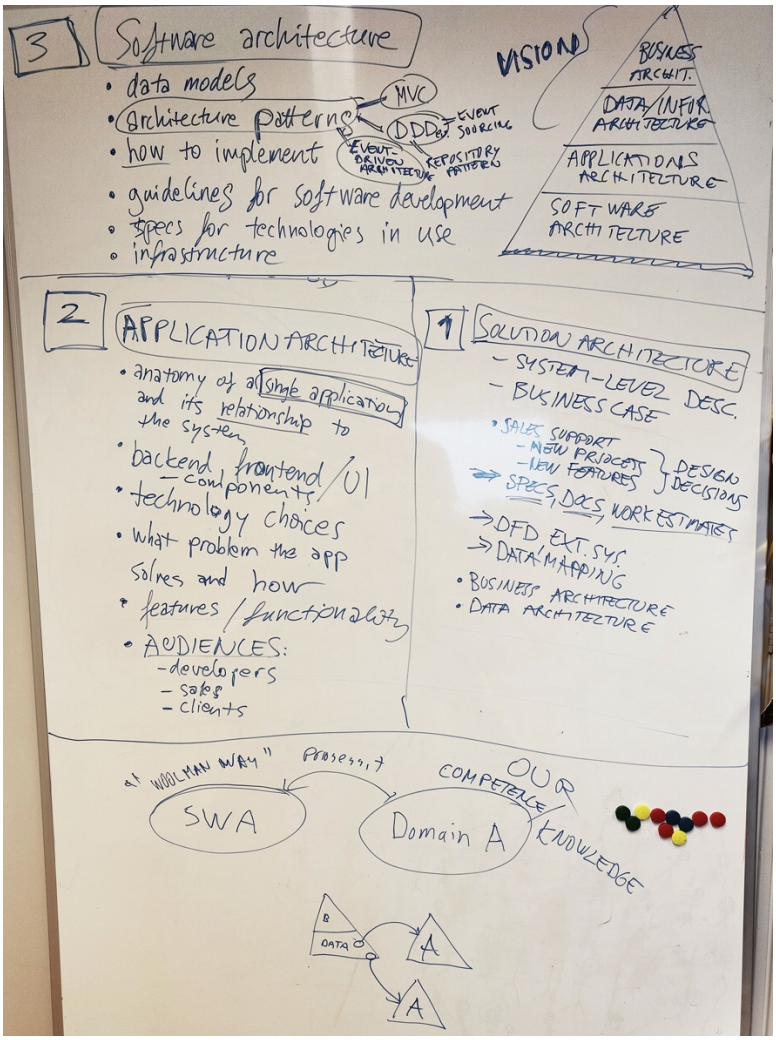


Figure 3. Architecture domains

I realize the value in this picture. It really helps us to align and communicate our work to the rest of the company. Here is the transcript:

Solution architecture is also known as “How to solve your customers’ needs with apps and software”. This topic explains the domain we are working in, and what kind of common solutions our customers are looking for. Good solution architecture exposes how we can deliver as much value as possible with minimum effort – the key to good architecture! It also helps us to communicate the solutions to our customers with useful diagrams and process descriptions. Solution architecture includes (but is not restricted to)

- Business case and business architecture

- System level design decisions
- All external systems involved
- Information architecture and data mapping

Application architecture describes a component-level implementation of the solution, e.g. what technology, application and feature we are going to use when we are creating a solution for the business case. Application architecture answers to the question: how can we achieve whatever is required in the business case? In our context, this domain typically explains the anatomy of a Shopify application and its requirements for front-end and back-end (e.g. Liquid page templates, JavaScript snippets, proxy app or middleware and API description). Application architecture includes (but is not restricted to)

- Backend and UI components
- Technology choices
- Features and functionality

Software architecture explains how we do software development at Woolman. It contains general architecture decisions that we have made but which are not related to one specific customer. Software architecture explains our default tools, products and components. In this architecture domain we go into more detail about technology stacks, how to use them successfully, and how to create re-usable but evolving software. We also define how solution architecture is transferred into software. Software architecture includes (but is not restricted to)

- Data models
- Infrastructure
- Architecture patterns
- Guidelines for software development

3.1.5 Friday 21 February 2020: Viewpoints

Yesterday gave me a lot to think. I want to understand the relationship between different architecture views we identified. I also want to compare our approach to industry standards.

I am reading a book called "*Software Architect Bootcamp"* (Malveau & Mowbray 2004). It introduces the major schools of software architecture thought like *Zachman Framework* and *4+1 View Model*. However, my interest is drawn to a formal standard from the International Standards Organization (ISO). This architecture approach is called *Reference Model for Open Distributed Processing* (RM-ODP).

The ODP model defines five essential viewpoints for modelling architecture (Malveau & Mowbray 2004, 17):

1. Enterprise viewpoint
2. Information viewpoint
3. Computational viewpoint
4. Engineering viewpoint
5. Technology viewpoint

Every point of view is addressing the needs of a particular stakeholder in the system. The *enterprise* viewpoint defines the business case. The *information* viewpoint represents the data and processes on data. The *computational* viewpoint divides the system into software components that enable distribution. The *engineering* viewpoint exposes the mechanism of distribution in the system (e.g. generic software components). And finally, the technology viewpoint maps the engineering objects to technology selections and specific standards. (Malveau & Mowbrau 2004, 17-19.)

I want to make a thought experiment. I try to map our own architecture approach to this ISO standard. My first diagram looks like this (Figure 4).

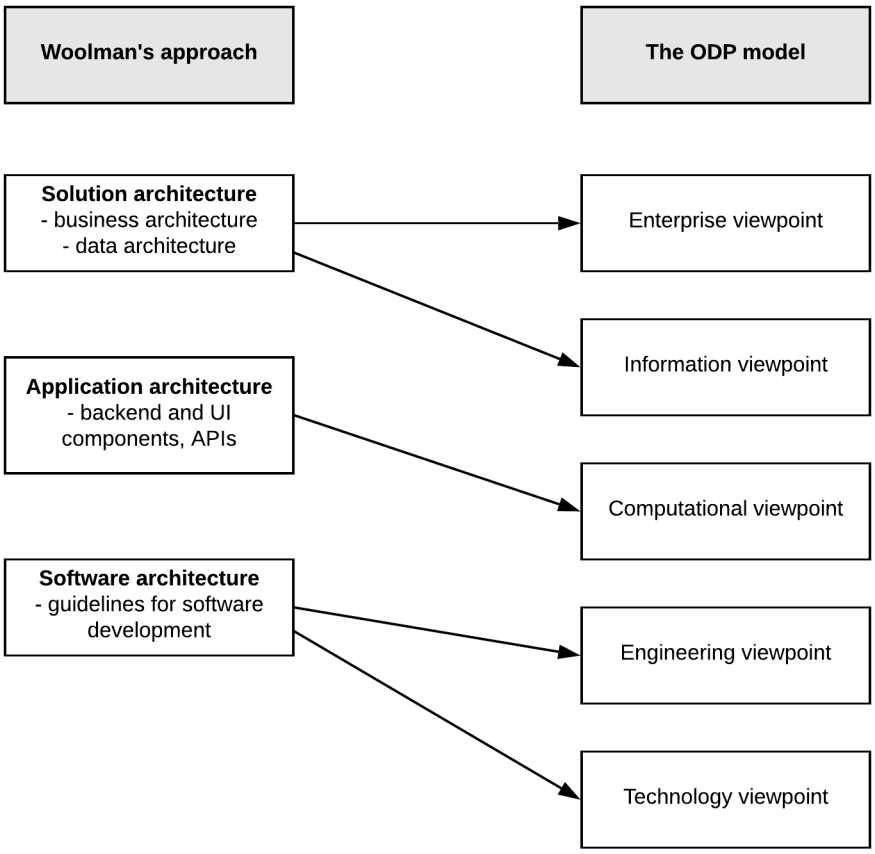


Figure 4. Architecture viewpoints

However, I realize that this diagram is misleading. Firstly, the authors emphasize that the five perspectives are not hierarchical but rather co-equal and complementary: (ibid., 19.)

"Each defines various constraints on the design of the information system that provide various architectural benefits for each of the system's stakeholder."

Secondly, our own architectural approach is not hierarchical either. In our minds, the solution architecture focuses mainly on the business case, while software architecture defines how the former is transferred into good software. Different perspectives to the same subject, right? I try to make another diagram where I pay attention to this diversity of viewpoints.

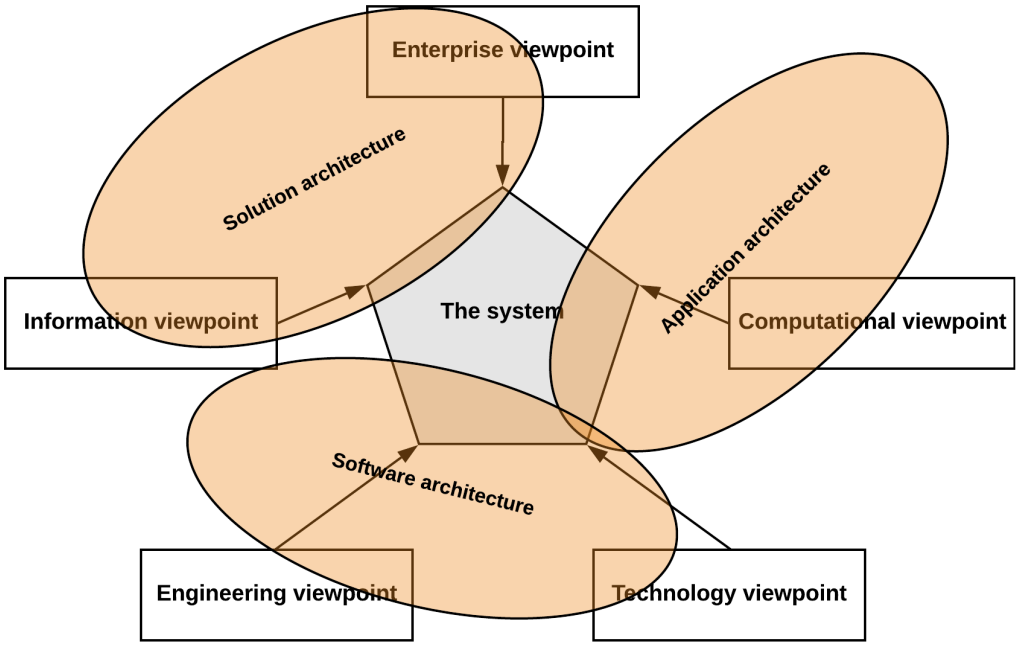


Figure 5. Combining viewpoints and domains

The diagram (Figure 5) is not entirely satisfying but better than the first one. It addresses the plurality of perspectives that need to take into consideration when producing good architecture documentation. As a solution architect, I can identify target groups like business owners, engineers, and end-users.

3.1.6 Weekly Analysis

During the week I have been getting familiar with software architecture domain. I have also reflected my own intuition upon the existing discursion. I find this comparison method productive, so I will probably continue it in the future too. I have identified several questions with respect to documenting software architecture in our company:

- who is responsible for architecture documentation?
- what should be documented?
- when should you produce architecture documentation?
- why is it valuable to make this effort at all?
- where software architecture should be documented?
- how should it be documented?

I believe that finding answers to these questions is important not only for my professional growth but also for the company. In some sense it feels like I am doing an investigative journalism, and this thesis will be my “watchdog report”. Surprisingly, I find a meta-level interconnection between my questions and one of the most popular architectural frameworks called *Zachman framework*. This framework includes six viewpoints and five levels of design abstractions (Zachman 1996).

Table 1. Zachman framework (adapted from Zachman 1996)

	What	How	Where	Who	When	Why
Planner						
Owner						
Designer						
Builder						
Operator						

The framework graphic (in its most simplistic form) represents the intersection of different perspectives and questions in the system design process (see Table 1). After taking a deeper look at the framework I find similarities but also divergence to my own concerns. As an example, Zachman states that *who* is referring to overall business responsibility (who are the people that run the business), and *when* is referring to timing (when are the processes performed in the workflow).

I like this systems thinking approach and migrate it to my own context. As a solution architect, you should strive not only to understand the system but also to improve it. You are part of the business itself. Additionally, documenting software architecture is not a one-time/one-man effort. It is rather a multiphase flow that affects the organization widely. As a solution architect, you should consider the timing of your work performances and reduce the waste of waiting.

3.2 Week 2

3.2.1 Monday 24 February 2020: Data Flow Diagrams

Today I start to design systems architecture for a furniture company. We are going to implement an ecommerce store for them, and we need to make quite a few design decisions. I decide to start with a logical data flow diagram (DFD) because it is usually a good instrument for collaborating and communicating with the client about their business requirements. I begin listing the entities on top of the diagram. Each external entity has its own column. In this case the entities are

- customer
- Shopify store
- integration application
- brick-and-mortar
- warehouse
- supplier

I want to describe the business process from purchase to order notification in different scenarios. For this I need various symbols. I am using a software called Lucidchart for diagramming. It has a bunch of built-in flowchart shapes that I am going to utilize. Here is my “toolbox” for today (Figure 6):

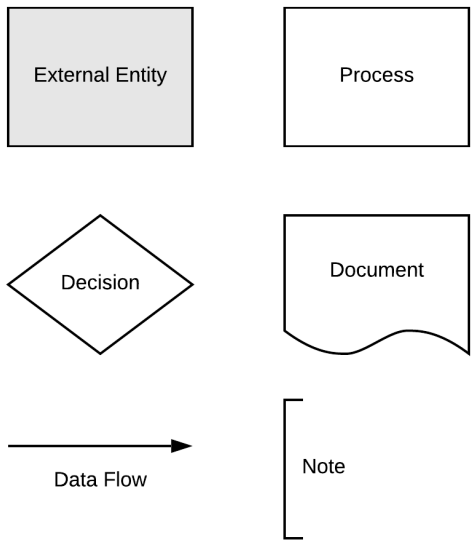


Figure 6. Flow diagram notation

The first draft looks like this (Figure 7):

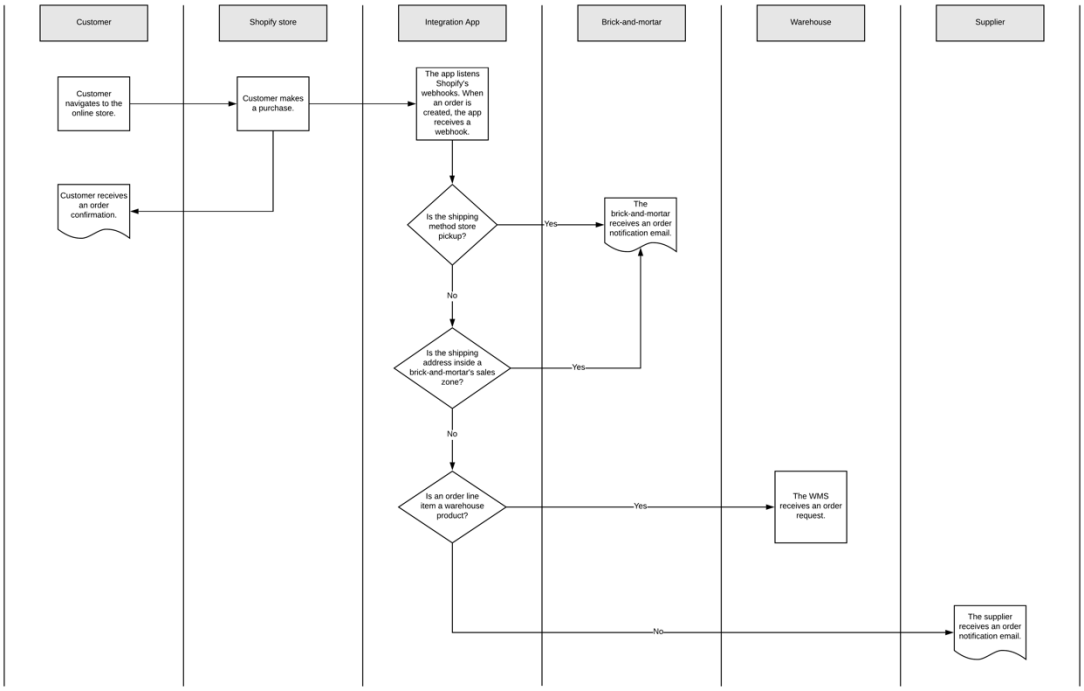


Figure 7. Flowchart

I decide to compare my own approach to standards. According to Lucidchart's documentation data flow diagrams usually include four main elements: entity, process, data store and data flow.

External […] entities produce and consume data that flows between the entity and the system being diagrammed. […] Since they are external to the system being analyzed, these entities are typically placed at the boundaries of the diagram. (Data Flow Diagram Symbols n.d.)

In my diagrams these entities usually represent systems/subsystems or operators (like end-users). I like to place them on top of the diagram in self-contained columns.

Process – An activity that changes or transforms data flows. Since they transform incoming data to outgoing data, all processes must have inputs and outputs on a DFD. (Data Flow Diagram Symbols)

I understand the notation in the same way. I usually write a short, descriptive title in the center of the box (like in this case “the WMS receives an order request”).

A data store does not generate any operations but simply holds data for later access. Data stores could consist of files held long term or a batch of documents stored briefly while they wait to be processed. (Data Flow Diagram Symbols)

I realize that my diagram does not have any data store in it. For me Shopify’s databases and APIs are something I consider self-evident. However, this might not be the case for our clients who are not familiar with the product. I need to take this in account in future.

Data Flow – Movement of data between external entities, processes and data stores is represented with an arrow symbol, which indicates the direction of flow. (Data Flow Diagram Symbols)

This is exactly how I use the notation in my own work too.

3.2.2 Tuesday 25 February 2020: Flowchart Symbols

I continue documenting the architecture for the furniture company. I am designing the fulfillment process and I need to include a few new flowchart symbols. After the additions my toolbox looks like this (Figure 8):

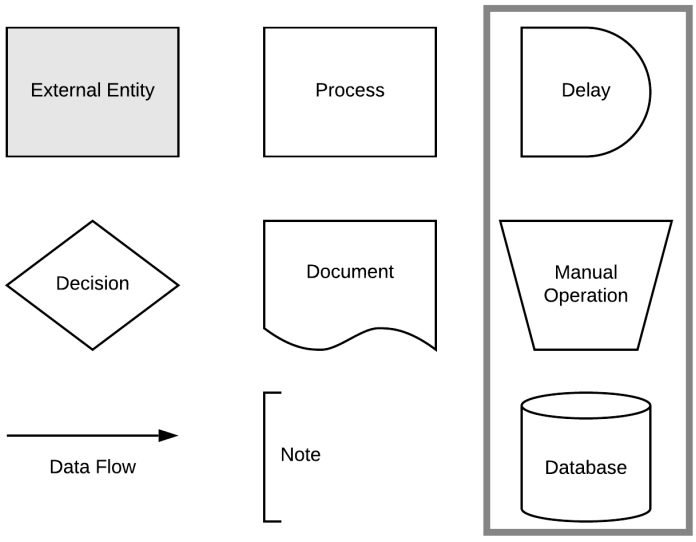


Figure 8. Additions to flow diagram notation

The first draft looks like this (Figure 9).

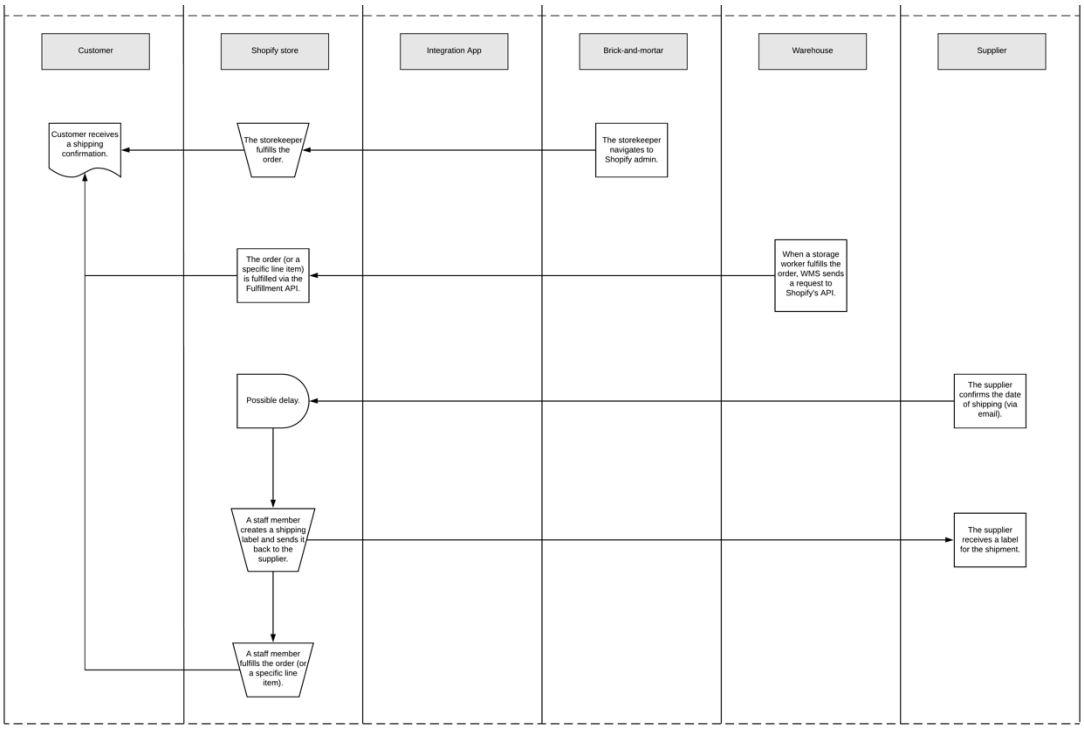
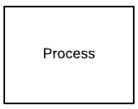
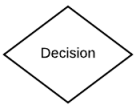
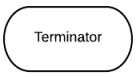
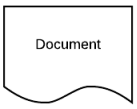
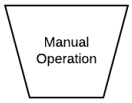
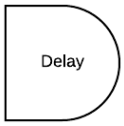
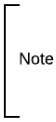
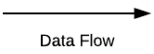


Figure 9. Second flowchart

I cannot help thinking that we should probably form a common practice for using symbols in our architecture diagrams. As an example, our personal styles to outline flow diagrams differ essentially. I discuss this with our integration architect, and he agrees. I decide to produce a baseline for further discussion. I concentrate on symbols and notation. The proposal suggestion follows our practical needs and best practices established by popular diagram software like Lucidchart (Flowchart Symbols and Notation) and Gliffy (Kaufman 2019). The table looks like this (Table 2):

Table 2. Definition of symbols (adapted from Flowchart Symbols and Notation)

Symbol	**Name**	**Description**
External Entity	External entity symbol	Represents a system, subsystem, or an actor within the diagram. Typically placed at the boundaries of the diagram.

	Process symbol	Represents a process, action, or function. This is the “workhorse” of the diagram.
	Decision symbol	Represents a question to be answered (e.g. yes/no or true/false). The path may then split off into different branches.
	Start/end symbol	Represents the start points and end points of a logical path.
	Document symbol	Represents the input or output of a document (e.g. receiving an email or a report).
	Database symbol	Represents data that will likely allow for searching and filtering by users (can be accessed in any order).
	Manual operation symbol	Represents a step that must be done manually, not automatically.
	Delay symbol	Represents a delay in the process. Defines a waiting period that’s part of the flow.
	Note symbol	Represents the needed explanation or comments within the specified section of the diagram.
	Data flow symbol	Represents the direction of the flow. Used also to guide the viewer along the path.

The table is not complete. It covers less than half of the flowchart symbols provided by our diagram software. However, these are the symbols I have been using lately, so it is a good start anyway.

3.2.3 Wednesday 26 February 2020: Diagram Keys

Today I am reading about "seven rules for sound documentation" introduced in the book called *"Documenting Software Architectures"* (Clements, Bachmann, Bass, Garlan, Ivers, Little, Merson, Nord & Stafford 2010, 36-45). The authors recommend using this checklist when writing technical documentation (ibid., 36):

1. Write documentation from the reader's point of view
2. Avoid unnecessary repetition
3. Avoid ambiguity
    a. Explain your notation
4. Use a standard organization
5. Record rationale
6. Keep documentation current but not too current
7. Review documentation for fitness of purpose

At the moment I am interested especially in rules 3 and 3.a. Clements and his colleagues write that "*ambiguity occurs when documentation can be interpreted in more than one way and at least one of those ways is incorrect"* (ibid., 40). Furthermore, the most dangerous one is *undetected* ambiguity. In other words, nobody even knows that each reader will come to a different conclusion. In fact, the authors give a clear advice: *"make sure you explain precisely what the boxes and lines mean"* (ibid.). How to achieve this?

According to Clements and others, the best way to do this is to include a key in diagrams (ibid., 41). This is the missing piece to my puzzle too. You never know who is going to read the architecture document ultimately because the timespan can be years. It can also be difficult to establish an all-embracing standard in a young company like ours. However, we could start to follow a practice like this and increase clarity by including a key to the symbology of the diagram. As the authors bring up in the margins (ibid., 40): *"every diagram […] should include a key that explains the meaning of every symbol used".*

In short, the key should identify the notation. Technically, I just have to bring together the diagram and the table of symbols and call the latter as a key. The outcome could be something like this (Figure 10):

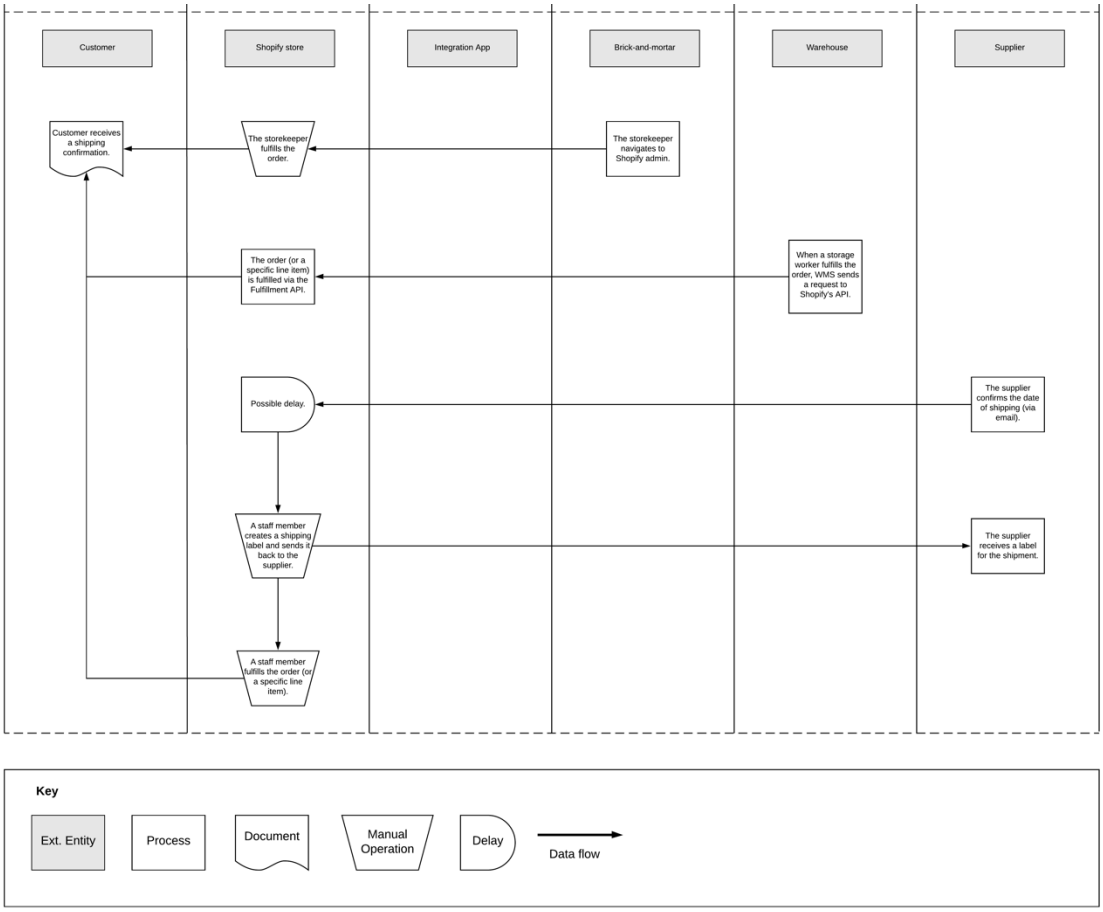


Figure 10. Diagram key

I realize I need to re-factor the diagram (Figure 10). As an example, according to the key “the supplier receives a label for the shipment” should be a document, not a process. Whatever happens after the supplier receives the document, is probably a process. Secondly, what do the arrows mean? A lot of things, I have to admit. As an example

- http REST
- an email is sent
- a logical path

I need to find a better solution for drawing this. I should probably use different arrow notation for each use case and include these in the key too. The first draft could be something like this (Figure 11):

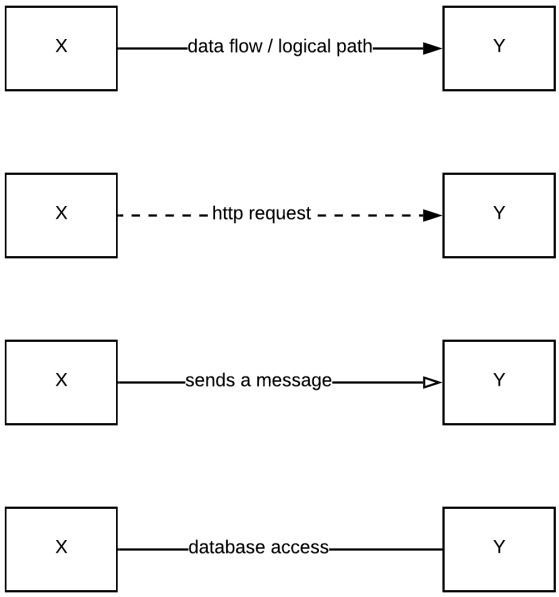


Figure 11. Arrow notation

3.2.4 Friday 28 February 2020: Documentation Process

Today I have been thinking the seventh rule for sound documentation: review documentation for fitness of purpose. Clements and others state that *"only the intended users of a document will be able to tell you whether it contains the right information presented in the right way"* (Clements, Bachmann, Bass, Garlan, Ivers, Little, Merson, Nord & Stafford 2010, 45). Architecture document should be reviewed by the stakeholders for which it was written. This practice should help us spot any obscurity.

I try to sketch a process chart for this. As we are developing a practice for documenting software architecture through empirical experiments, I consider the Plan-Do-Check-Adjust (PDCA) learning cycle as a good starting point. My approach to the PDCA cycle follows roughly the industry standard introduced by Scaled Agile Framework (Iterations). What would be the relevant steps in the context of architecture documentation? Since the document should be surrounded by discussion and editable, the PDCA steps can be mapped to the "Views and Beyond" philosophy introduced by Clements and colleagues (ibid., 19). Here is my idea of the mapping:

- Plan → finding out what stakeholders need.
- Do → recording design decisions.
- Check → checking the resulting documentation.
- Adjust → packaging the information in a useful form.

The authors make it very clear that Views and Beyond "*does not have a sequence of steps, with entry and exit criteria for each"* (Clements et al. 2010, 19). Rather the premise of their philosophy is that documentation *"should be the helpful result of making an architecture decision, not a separate step in the architecture process"* (ibid., 19-20). In my opinion this maxim goes really well with the PDCA learning cycle – even though it does not address any timebox for the loop (which is the presumption in the PDCA). As a result, I get this kind of a process chart (Figure 12).

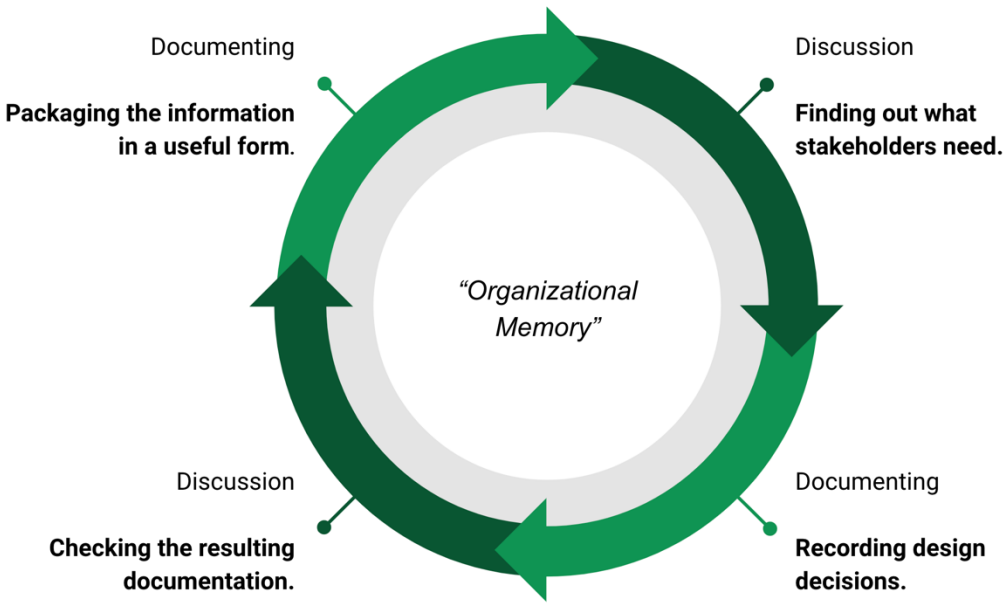


Figure 12. Organizational memory

I like the idea of putting organizational memory in the heart of the diagram (Figure 12). It is not originally present in Views and Beyond philosophy or the PDCA cycle. However, the concept is highly meaningful in our business context. Organizational memory can be seen as a collective ability to store and retrieve knowledge and information (Organizational Memory and Knowledge Repositories). Each time a

significant design decision is made, some information should be added to the organizational memory. To this end, architecture documentation is a fundamental building block.

3.2.5 Weekly Analysis

I have read through the diary entries I have made so far. Mixing theory and practice is working well for me and I have already learned a lot. Nevertheless, the emerging discourse of software architecture is somewhat abstruse to me. As an example, I have used all these concepts so far (not listed in any specific order):

- architecture documentation
- document repository
- system
- viewpoint
- view
- stakeholder
- perspective
- architectural benefit
- architectural framework

I am not interested in formal concept analysis. It would be a huge effort and not the priority of my interests (which are rather business oriented). I am applying myself to developing a *practice* for documenting software architecture in our company. However, some kind of a consistency is necessary – or I might lose myself in jargon. Most of all, I want to understand how these frequent concepts are related to each other. I have to look more into this.

3.3 Week 3

3.3.1 Monday 2 March 2020: ISO 42010 Standard

Today I have looked for more details of frequently used concepts in software architecture literature. I have become aware of that the International Organization for Standardization has more than one standard for software architecture. ISO 42010 addresses *“the creation, analysis and sustainment of architectures of systems through the use of architecture descriptions”* (Systems and software engineering — Architecture description 2011a).

Only informative sections of ISO standards seem to be publicly available. However, the introduction chapter indicates that ISO 42010 can be used to *"establish a coherent practice for developing architecture descriptions, architecture frameworks and architecture description languages within the context of a life cycle and its processes"* (Systems and software engineering — Architecture description 2011b).

According to Maier and Rechtin (2009, 325), this ISO standard codifies the structure of an architecture description independently of any specific architecture framework (which I consider as a good thing because it gives you more "elbowroom").

In the […] ontology, every system has one architecture. That architecture can have several architecture descriptions. […] An architecture description is composed of stakeholders, concerns, viewpoints, views, and models. […] Viewpoints may be drawn from a viewpoint library. (ibid., 325-326.)

Gratifyingly, the authors have made an informational model of the core concepts (ibid., 325). The diagram is written in Unified Modeling Language (UML), which I am not an expert on, but it can be easily interpreted anyhow. This (Figure 13) is an adapted, non-UML version of that very diagram:

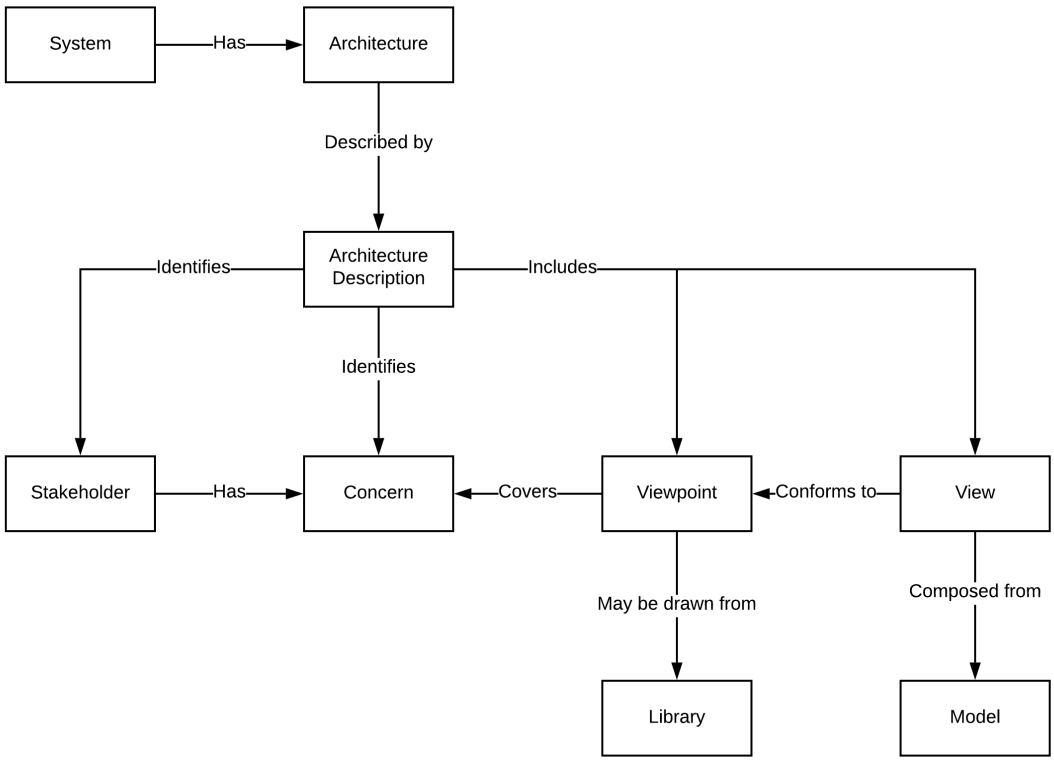


Figure 13. ISO 42010 overview (adapted from Maier & Rechtin 2009, 325)

I find this (Figure 13) very useful because it illustrates the relations of different concepts in a compact form. I also become interested in an idea that viewpoints may be placed in a library for later use. This could benefit our business and reduce re-work in the long run. However, the simplified diagram also raises questions. What are the underlying tenets behind ISO 42010 and its terms?

3.3.2 Tuesday 3 March 2020: ISO 42010 Standard

Today I am studying the glossary defined by ISO 42010 standard. I have created a mapping table (Table 3) between a term and its definition (Systems and software engineering — Architecture description 2011b).

Table 3. Definition of ISO 42010 terms (adapted from Systems and software engineering — Architecture description 2011b)

Term	**Definition**
architecture description	work product used to express an architecture
architecture framework	conventions, principles and practices for the description of architectures established within a specific domain and/or community of stakeholders
architecture view	work product expressing the architecture of a system from the perspective of specific system concerns
architecture viewpoint	work product establishing the conventions for the construction, interpretation and use of architecture views to frame specific system concerns
concern	interest in a system relevant to one or more of its stakeholders
model kind	conventions for a type of modelling (e.g. data flow diagrams)
stakeholder	individual, team, organization, or classes thereof, having an interest in a system

According to Clements and others (Clements, Bachmann, Bass, Garlan, Ivers, Little, Merson, Nord & Stafford 2010, 400-401), the standard of ISO 42010 is focused on two key ideas: a conceptual framework for architecture description, and a statement of what information must be found in that description. The standard requires the following (ibid., 402):

- Identification of the stakeholders
- Identification of the (architecture-related) concerns of those stakeholders
- A set of architecture viewpoints defined so that all of those concerns are covered
- A set of architecture views (one view for each viewpoint)

For me, the first two bullets are easy to understand. However, I am not sure why you must separate a view and a viewpoint. In the diagram, a view conforms to a viewpoint. Makes sense, but what value the viewpoint adds to the view? As can be

seen from the table, a view expresses the architecture of a system from the perspective of specific concerns. If the concerns are already covered by the view, why you need to define a viewpoint in the first place?

According to Clements and colleagues (2010, 402), it is useful to separate viewpoints (*perspectives on the architecture*) from views (*what is captured in the architecture description*) because a viewpoint explains the conventions being used in that view. Furthermore, viewpoints are selected for use to ensure coverage of the identified concerns. In this sense, a viewpoint comes before the view (not *vice versa*).

I think that the key to understanding this is already in the table: both a view and a viewpoint are *work products* – concrete artifacts made for repeated use. So, the logical question is: *how does these artifacts look like?* Views are easier to understand. They are more often graphical presentations with "boxes and lines" – like the diagrams I have been drawing while writing this paper. However, viewpoints are mysterious to me. What do viewpoints look like and how do I write them?

Fortunately, Richard Hilliard has made a template for specifying architecture viewpoints in accordance with ISO 42010 (Hilliard 2012). The basic idea is to tell the reader in plain language

- who are the typical stakeholders?
- what are their concerns?
- what notations, models and techniques will you use (as an architect)?

Eventually, this re-usable document becomes a contract between the architect and stakeholders that *"these concerns will be addressed in the view resulting from this viewpoint"* (Hilliard 2012, 4). Moreover, Hilliard gives tips how to use the template he has provided. As an example, it can be helpful to express concerns in the form of questions:

- How does the system handle network latency?
- How does the system manage faults?
- What services does the system provide?

3.3.3 Wednesday 4 March 2020: Viewpoints

Today I have been thinking the abstraction of an architecture viewpoint. At the moment we are not utilizing the concept in our work. Should we? I am not sure. It would mean extra work for sure. According to ISO 42010, we would need to document the viewpoint in addition to the view itself. On the other hand, I like the idea that viewpoints could be re-usable and stored in our document library for later use. Some extra work today might pay back tomorrow. It also could be an instrument for quality assurance.

As stated by Malveau and Mowbray (2004, 11), predefined viewpoints have the advantage that they can accompany a well-defined terminology for resolving consistency among different sets of clients and projects (and architects too!). In an ideal world, we would have a comprehensive library from among we could select a group of “pressure tested” viewpoints for specification.

Frankly speaking, the current state of our architecture work is rather unformed. In a typical ecommerce project, the concerns of stakeholders are scattered in notebooks, emails, business proposals, Slack messages and so forth. Secondly, these concerns are comparable by nature from one project to another. So, this might be a good chance of being reproducible.

As an example, a marketing manager needs to track his/her online campaigns. Bookkeeper needs to see taxes report, while storage worker has to print shipping labels. Customer in turn wants to track the shipment and get delivery status online. Product manager wants to know which products are the most profitable and which he/she should re-order when. Staff member must know how to make a refund after receiving a reclamation, and so on. On average, these concerns are predictable.

Maybe we should document these recurrent concerns of stakeholders as viewpoints and ensure that they are systematically covered in our projects. This would probably prevent design errors and improve customer experience. In its lightest form, a viewpoint could be a “checklist” that guides the architect in his search for the best possible solution.

3.3.4 Thursday 5 March 2020: Strategic Design

I begin to understand how extensive theme architecture is. I did not realize the big picture before, even though it was right in front of my eyes. I am referring to this picture (Figure 14) we draw earlier.

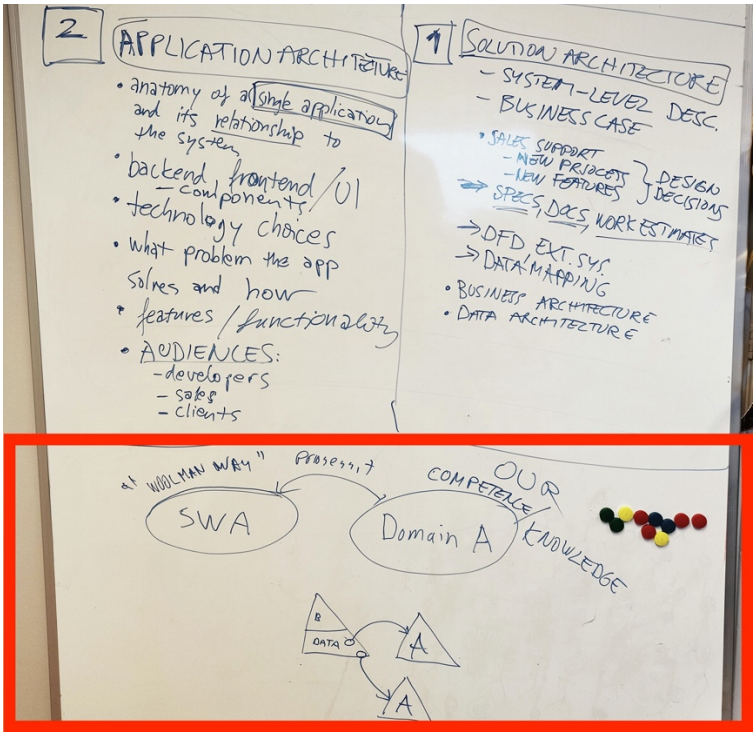


Figure 14. Architecture domains and strategic design

My first impression was that solution architecture, application architecture and software architecture are somehow hierarchical, but I quickly turned away from this thought. However, I did not realize the sophisticated meaning hiding at the bottom of the picture. Our senior architect tried to illustrate the role of architectural design in a more strategic manner. I believe he meant something like this (Figure 15).

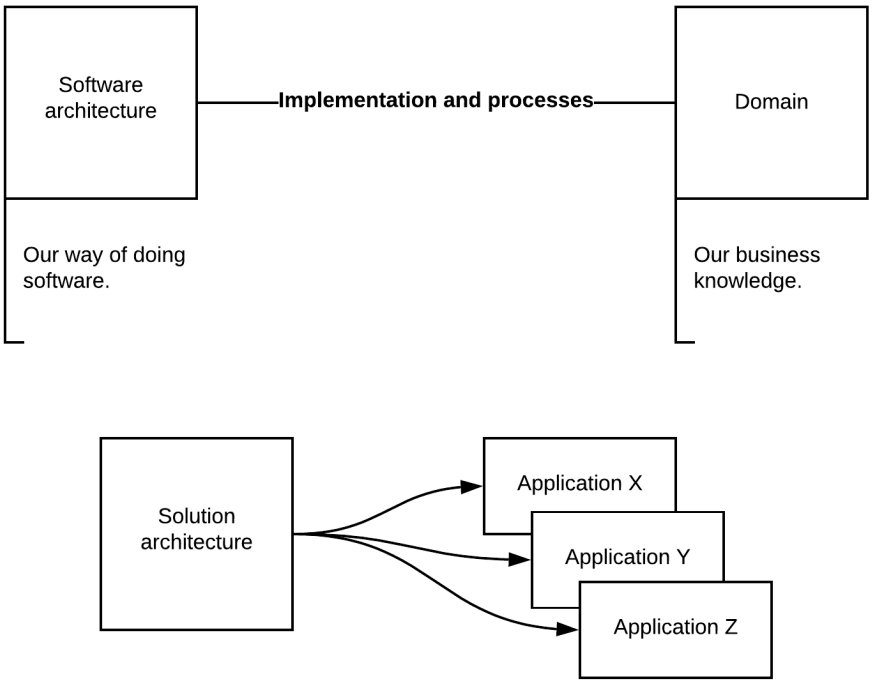


Figure 15. Software architecture and business domains

How should we interpret this diagram (Figure 15)? My colleague gave me a hint and encouraged to read up on domain-driven design (DDD). According to the community (What is Domain-Driven Design?), DDD provides practices and terminology for making design decisions with complicated business domains. Furthermore, *"the most significant complexity of many applications is not technical. It is in the domain itself, the activity or business of the user."*

In other words, the key to the diagram is the profound connection between software and business. While I was thinking that how we should document the design decisions rightly, my colleague was probably thinking that how we can make right decisions in the first place. According to Vaughn Vernon (2013, 7), strategic design helps us to understand what are the most important software investments to make. In this sense, DDD is more concerned with the strategic direction of the business (ibid., 9). To be able to understand this architectural approach, I need to define the meaning of a domain and its role in strategic design.

3.3.5 Friday 6 March 2020: Domain-Driven Design

Today I will continue studying domain-driven design. According to Vernon (2013, 43), a domain is *"what an organization does and the world it does it in".* As an example, when you develop software for an organization, you are working in its domain (ibid., 44). Our mission at Woolman is to help European brands succeed in global commerce. As a team, our competence is most closely related to ecommerce. In short, that is the domain of our business.

According to Vernon, separating distinct areas of the business domain will help us succeed. How is that exactly? The author states that any attempt to define the business in a single, all-encompassing (enterprise) model will be extremely difficult and probably fail. You should rather think about each of those business functions separately as subdomains. (Vernon 2013, 44.)

Let's make a thought experiment and outline what this means in our context. If the domain is ecommerce, what are the subdomains? One way to analyze this is to list common concerns of our stakeholders (I already wrote about this in chapter 3.3.3). As an example, a marketing manager needs to track his/her *online campaigns*. Bookkeeper needs to see *taxes report*, while storage worker has to print *shipping labels*. Customer in turn wants to track the shipment and get *delivery status* online. Product manager wants to know *which products are the most profitable* and which he/she should re-order when. Staff member must know how to make a *refund* after receiving a *reclamation*, and so on. As we can see, these concerns are all related to each other (see Figure 16).

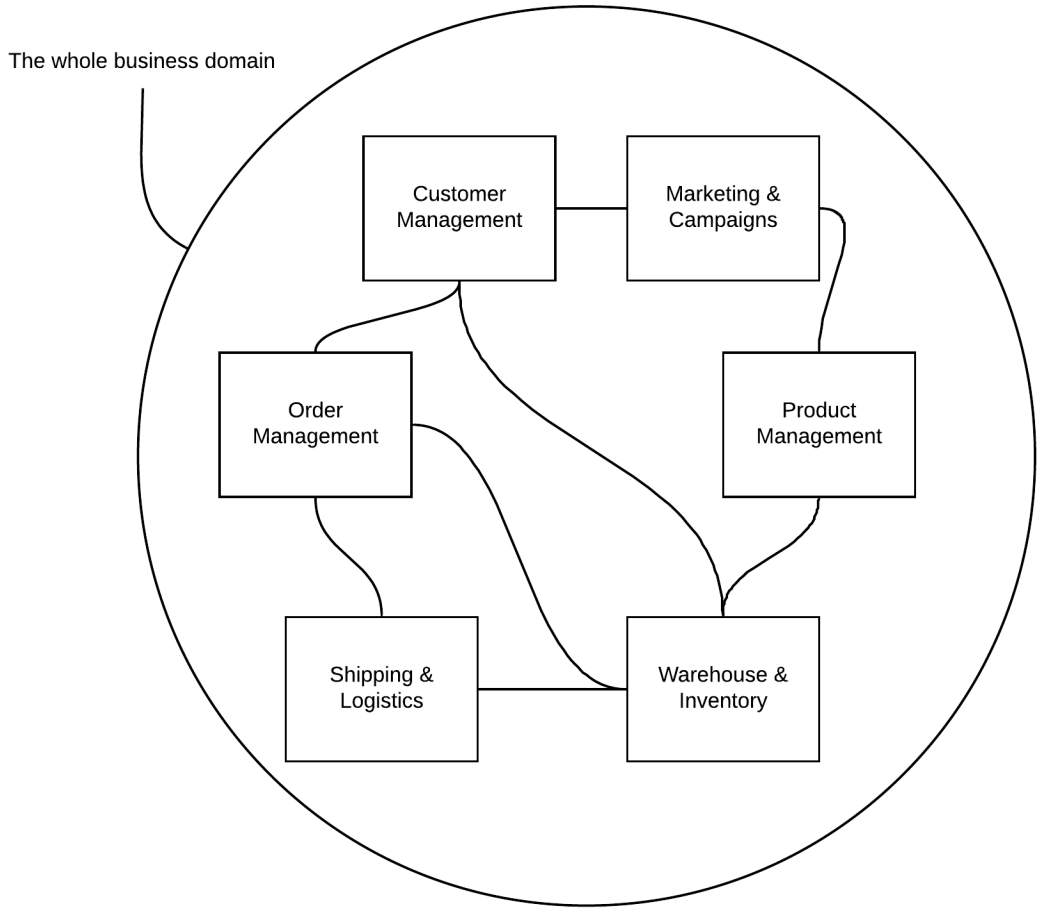


Figure 16. Domain-driven design

A typical process flow for an online business could be described like this:

1. Product marketing affects customers, who make orders.
2. After receiving orders, products must be shipped from the warehouse to the buyers.

Of course, this representation is way too simple. As an example, any retail company would need to collect payments for the orders. These payments must be recorded in accounting. Additionally, a retail company might have several online stores in different countries and a few brick-and-mortars too. Maybe they are also selling on Amazon and using fulfillment centers. Did I already mention information security and consumer protection?

In fact, we had a session with the architect team where we tried to identify the relevant subdomains altogether. After a few iterations, we ended up in this diagram (Figure 17).

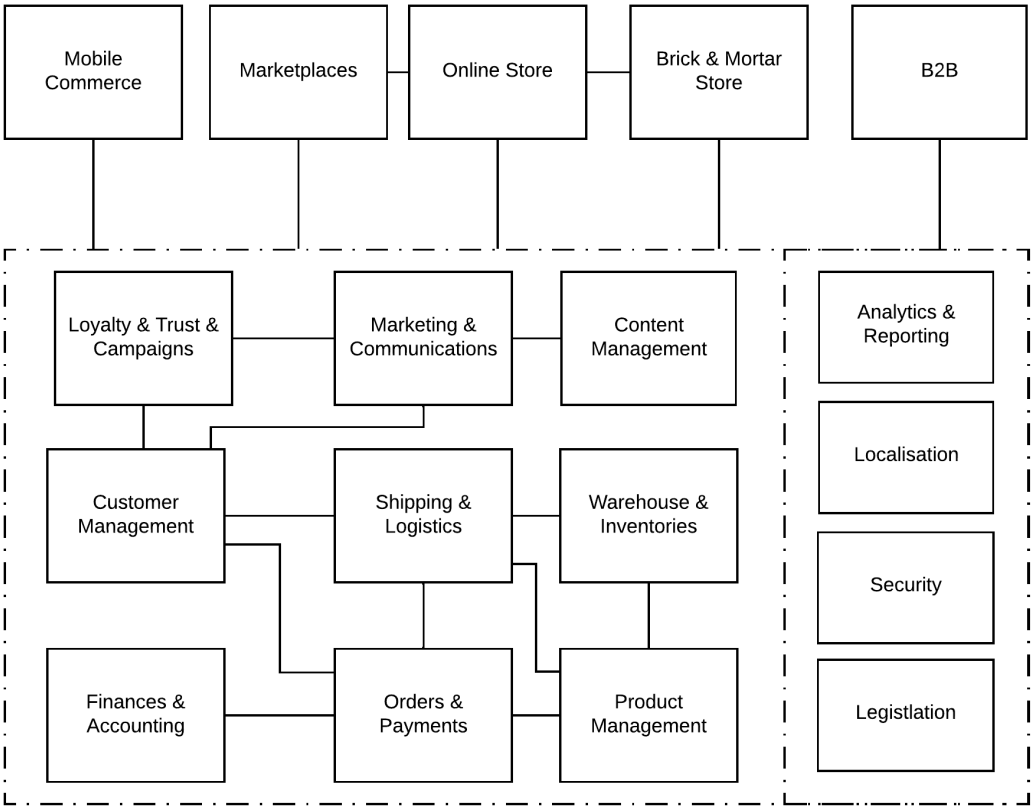


Figure 17. Ecommerce domain

It seems reasonable to separate customer facing sales channels (top row) from the backend operations. As can be seen, marketplaces (like Amazon) and brick-and-mortars are closely related to online stores but wholesale is a more independent area. This is not always the case, but usually wholesale buyers' needs differentiate essentially from consumer sales (volume of goods, customer-specific price lists, B2B payment methods, user management etc.). The right edge can be interpreted as nonfunctional requirements (NFRs) that define system quality like security and legality (Nonfunctional Requirements). Analytics and reporting are covering all subdomains by default.

However, according to Vernon (2013, 43) this kind of a “domain diagram” is just the beginning. To be able to learn the big picture of DDD and the foundation of strategic design, I must also make sense of *Bounded Contexts*. I try to cover this topic in the upcoming diary entries.

3.3.6 Weekly Analysis

This week has been eye-opening. I have followed my thoughts from ISO standards to architecture viewpoints and beyond. I understand better the discourse of software architecture. I have detected a shift from “tactical” design considerations to more “strategic” approach. At the moment it feels like architecture is bridging business strategy to execution (and vice versa). It binds together subdomains, viewpoints, and systems, resulting in a repository of knowledge. It will be interesting to see what domain-driven design has to offer in this sense. This is the learning path I want to follow next.

However, I am also aware of that I need to develop my competence of documenting architecture in practice. According to Paul Clements and others (2010, 45), this is a matter of documenting the relevant views, and then adding a binding documentation that applies to all of the views. The authors state that as an architect, you need to think about the software in three ways: “*Plan for your documentation package to include at least one module view, at least one component-and-connector view, and at least one allocation view”* (ibid., 49). After studying domain-driven design and bounded contexts, I will look in to these three categories of architecture styles.

3.4 Week 4

3.4.1 Monday 9 March 2020: Bounded Contexts

According to Martin Fowler (2014), Bounded Context is a central pattern in the strategic section of domain-driven design (DDD). He states that DDD is about designing software based on *models* of the underlying domain. A model has two functions. Firstly, it helps communication between developers and (business) domain

experts. Secondly, it acts as the conceptual foundation for the architecture design. Furthermore, to be effective a model needs to be internally consistent.

"As you try to model a larger domain, it gets progressively harder to build a single unified model. Different groups of people will use subtly different vocabularies in different parts of a large organization. The precision of modeling rapidly runs into this, often leading to a lot of confusion." (Fowler 2014.)

Actually, Vernon emphasizes the very same problem. According to him, it is almost impossible to specify concepts of a domain in a way that they have a single, pure, and distinct meaning among all stakeholders (Vernon 2013, 62).

"Some projects fall into the trap of attempting to create an all-inclusive model, one where the goal is to get the entire organization to agree on concepts with names that have only one global meaning. Approaching a modeling effort in this way is a pitfall." (Vernon 2013, 62.)

Frankly speaking, this is quite surprising. Why are these veteran software architects so worried about *linguistics*? And what it has to do with documenting an architecture? I think that their underlying motive is to prevent misunderstanding that would lead to a bad design – and poor software. As an architect, you should know exactly the meaning of the concepts that you use in your work. Furthermore, you should understand that same concepts have various meanings in different contexts.

So, how to solve this linguistic jumble lurking in large and complex systems? According to Vernon (2013, 62), the best practice is to apply Bounded Context to separately outline each domain model. As a result, inside each boundary all terms have specific meaning, and differences between models are well understood and documented. In other words, DDD's strategic design *"goes on to describe a variety of ways that you have relationships between Bounded Contexts"* (Fowler 2014). This kind of a concept mapping should defend your systems against painful design errors.

3.4.2 Tuesday 10 March 2020: Bounded Contexts

Today I want to apply Bounded Context. I am interested in seeing what *practical value* DDD brings to architecture design, if any. I have been thinking that what would be a good target for the experiment. I decide to concentrate on customer

management because it has been bothering me lately. We have a client that sells books, office supplies and games online. They have one million loyalty members, so the scale is huge. This sounds silly, but the problem is the definition of a customer. *Who is the customer?*

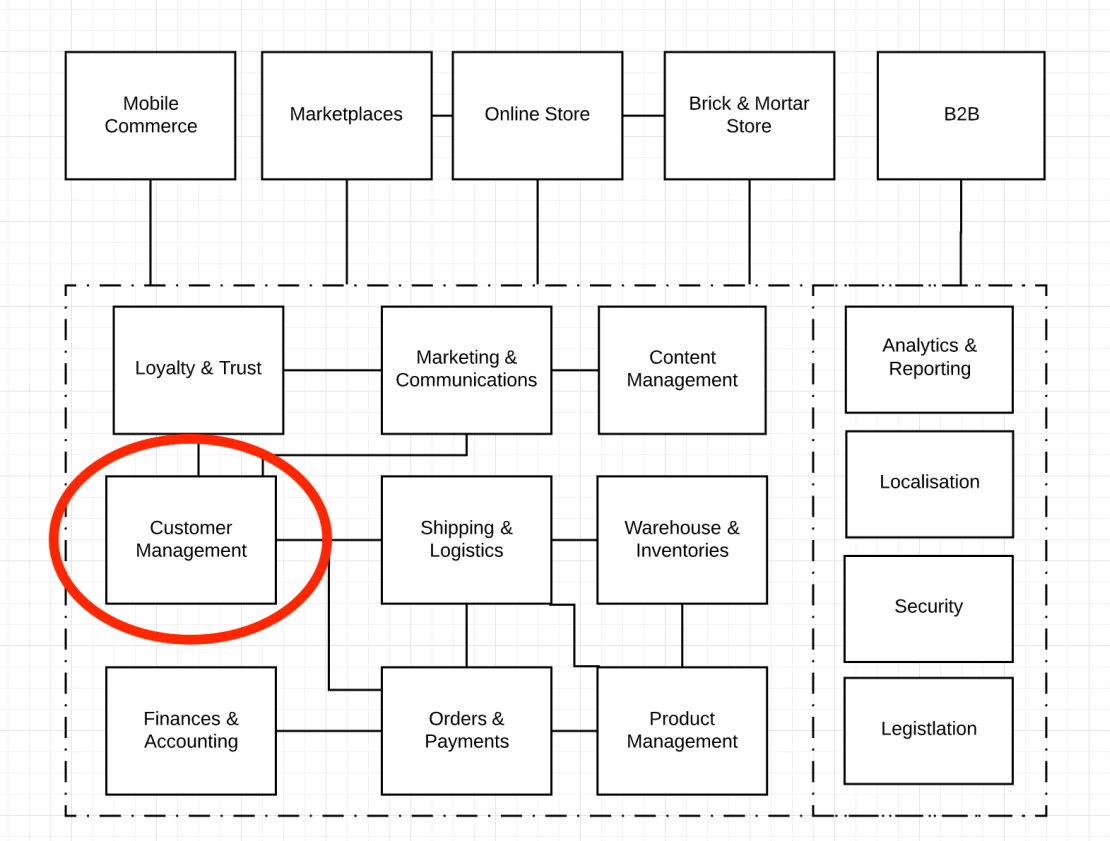


Figure 18. Customer management as an ecommerce subdomain

At first glance, the subdomain is explicit (see Figure 18). Customer management is linked to orders, shipping, marketing and loyalty program. Furthermore, a customer is a living creature – she has a name and a home (hopefully). However, from the architect's point of view, this obvious definition quickly runs into problems. Usually every customer needs to be identified. As an example, if you want to have a bank account or buy a new house, you probably have to show your ID to someone at some point. In our domain, social security number is not an option for identification or authentication. What is it then? It depends on the context.

Let’s say that you make an order online as a new customer. The ecommerce platform automatically generates a unique identifier for you. This customer id must be used whenever retrieving or updating your customer record via REST Admin API (for more information: https://shopify.dev/docs/admin-api/rest). Furthermore, as a customer you can only checkout using email. This email is unique too: attempting to assign the same email address to multiple customers would return an error.

On the other hand, an email address as an identifier only makes sense in the context of the online store. If you are making a purchase in brick-and-mortar, no one is really interested in your email. In fact, you do not even need to have one. If you want to collect purchase bonuses, you must identify yourself with a loyalty card. In this context, your loyalty program number is your identifier.

Both sales channels (online store and brick-and-mortar) are integrated to the company’s CRM solution, which is also the main platform for marketing activities such as email marketing. CRM has its own unique identifier for every customer record. It is called “external id”, and it consist of letters, numbers and dashes. Additionally, all these systems are connected to the legacy ERP system which has its own customer number space for each customer group (B2C, B2B etc.). So far, we have identified five different customer identifications:

- customer id (online store)
- email address
- loyalty card number (point of sale)
- external id (CRM)
- customer number (ERP)

According to our diagram, customer management is connected to the subdomain of shipping and logistics too. What would be the correct identifier in this context? None of the previous ones would make any sense. Carriers are only interested in shipping address and phone number (if they need to notify the customer about the shipment). How to design systems architecture without any contradictions in a complex domain like this?

3.4.3 Wednesday 11 March 2020: Bounded Contexts

Today I am going to continue my experiment and delineate a logical boundary around each context in the subdomain of customer management. According to Vaughn Vernon (2013, 90), I should start by drawing a *simple diagram* of the current situation that communicates at high level where the boundaries are, and the relationships between them. In other words, I should not overdo it but rather avoid ceremony and remain agile. I start with the online store and proceed quickly to the other contexts. I end up in a diagram like this (which is not perfect but shows the way).

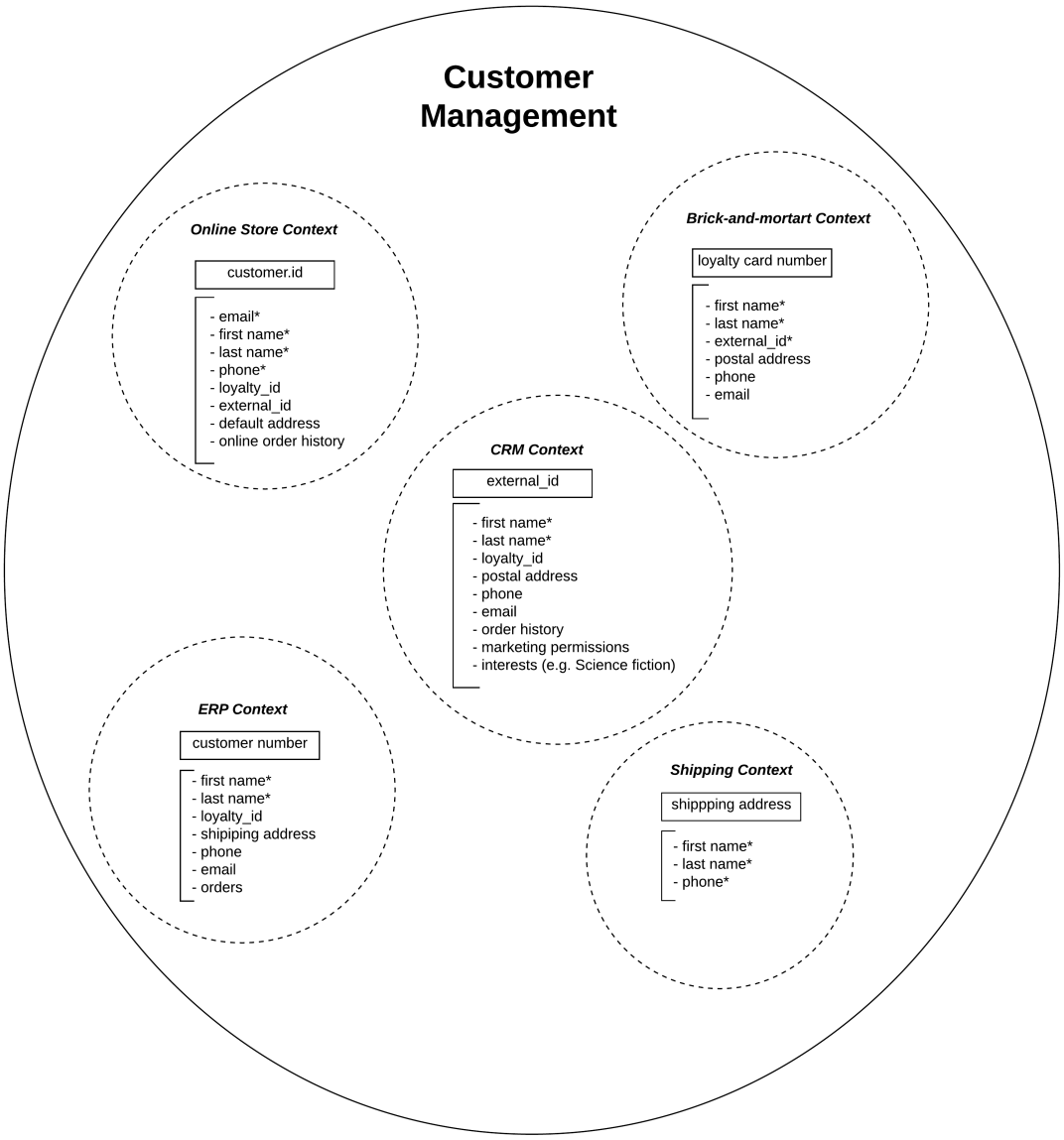


Figure 19. Bounded contexts

Now that I have identified the Bounded Contexts, and I am able to see them all in the diagram (Figure 19), it is easy to agree with Martin Fowler. He states that Bounded Contexts have both unrelated concepts (such as customer interest only existing in a CRM context) but also shared concepts (such as names and addresses). Moreover, different contexts may have completely different mechanisms to map between these polysemic concepts for systems integration. (Fowler 2014.)

3.4.4 Thursday 12 March 2020: Bounded Contexts

I keep on working with the experiment. What would be the next step? According to Vernon (2013, 73), in addition to subdomains and Bounded Contexts, I should grasp context mapping with integrations. I decide to draw the relationships of the five Bounded Contexts.

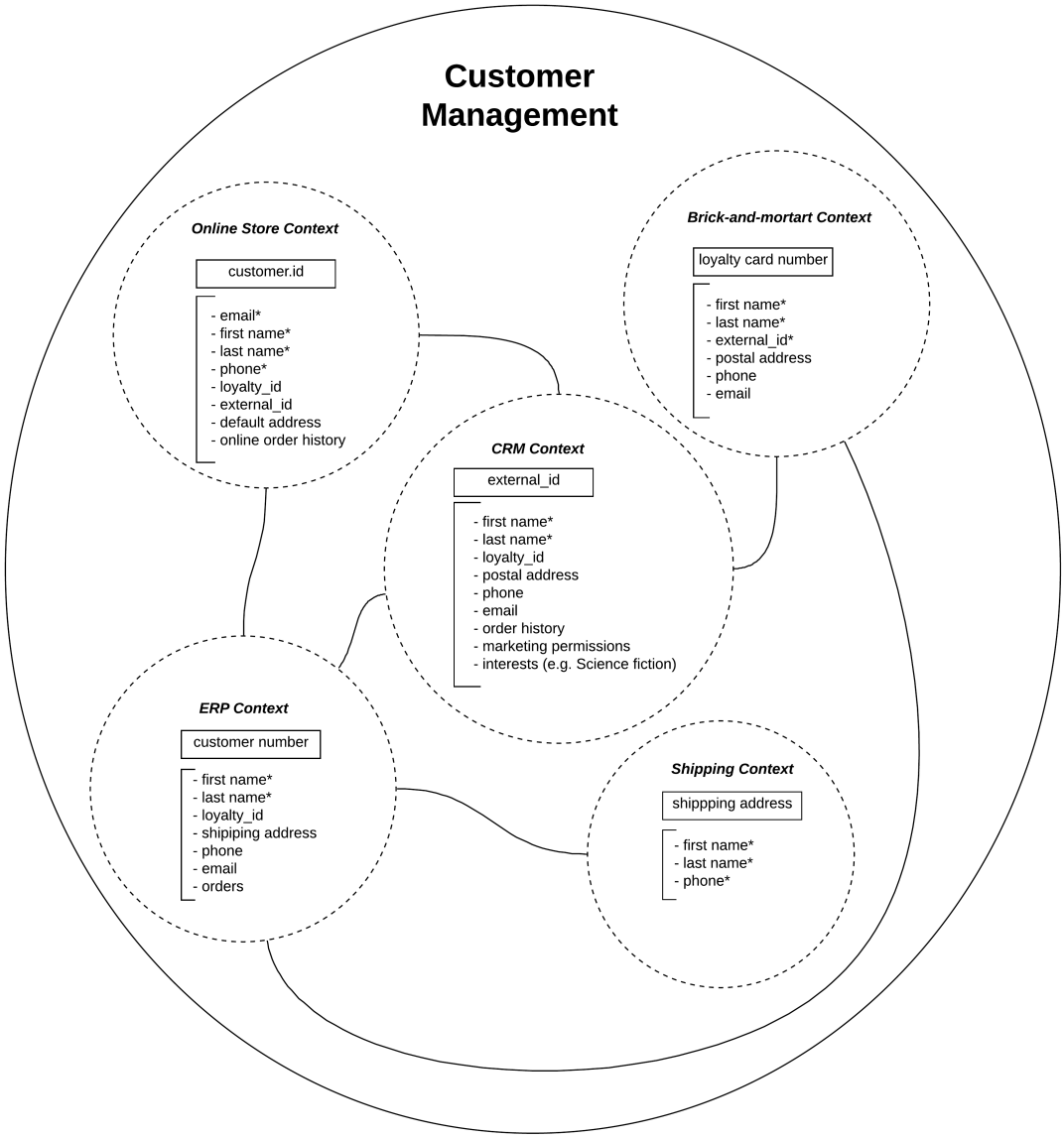


Figure 20. Integrating Bounded Contexts

As we can see (Figure 20), from the online store's perspective we have two major integrations: one between the online store and CRM, and another between the

online store and ERP system. Vernon states that in DDD, context maps have two primary forms: *"One form is a simple drawing that is used to illustrate the kinds of relationships that exist between any two or more Bounded Contexts. The second and far more concrete form is the code that actually implements those relationships."* (Vernon 2013, 449.)

I am interested in seeing how this "concrete form" would look like. According to Vernon (2014, 450), when Bounded Contexts need to integrate, there are a few reasonably straightforward ways this can be done:

- Use of a remotely accessible application programming interface (API). The API could be made available using SOAP or support sending XML requests and responses (not the same as REST).
- Use of a message queue. These messaging gateways can be thought of as service interfaces.
- Use of RESTful HTTP which means exchanging and modifying resources that are uniquely identified using a distinct URI. Various operations can be performed on each resource (e.g. GET, PUT, POST, DELETE).
- Use of other means of integration such as file-based or shared-database integration.

In this case, the last two bullets are the most relevant. Let's say that you make an order online as a new customer. We need to transfer the order from the online store to the ERP system. Within the context of the legacy ERP, our integration application listens ecommerce platform's webhook events, then converts the order JSON payload to the ERP-specific XML format, and finally uploads the order XML file to the SFTP server. During that conversion, the integration application generates and assigns a unique ERP-specific customer number to you from the pre-defined number space. At this point, you already have three identifiers:

- customer id (generated by the ecommerce platform)
- email address (provided by you during the checkout process)
- ERP customer number (generated by the integration application)

The next example is even more interesting. Your order details are transmitted from the SFTP server to the ERP system and data warehouse. Furthermore, the data is migrated to the CRM system. In this phase, you get the fourth identifier which is an external id. Let's imagine that you return to the online store the following day. Now

you want to register online and sign up to the loyalty program to get better deals in future. The process is somewhat like this (Figure 21):

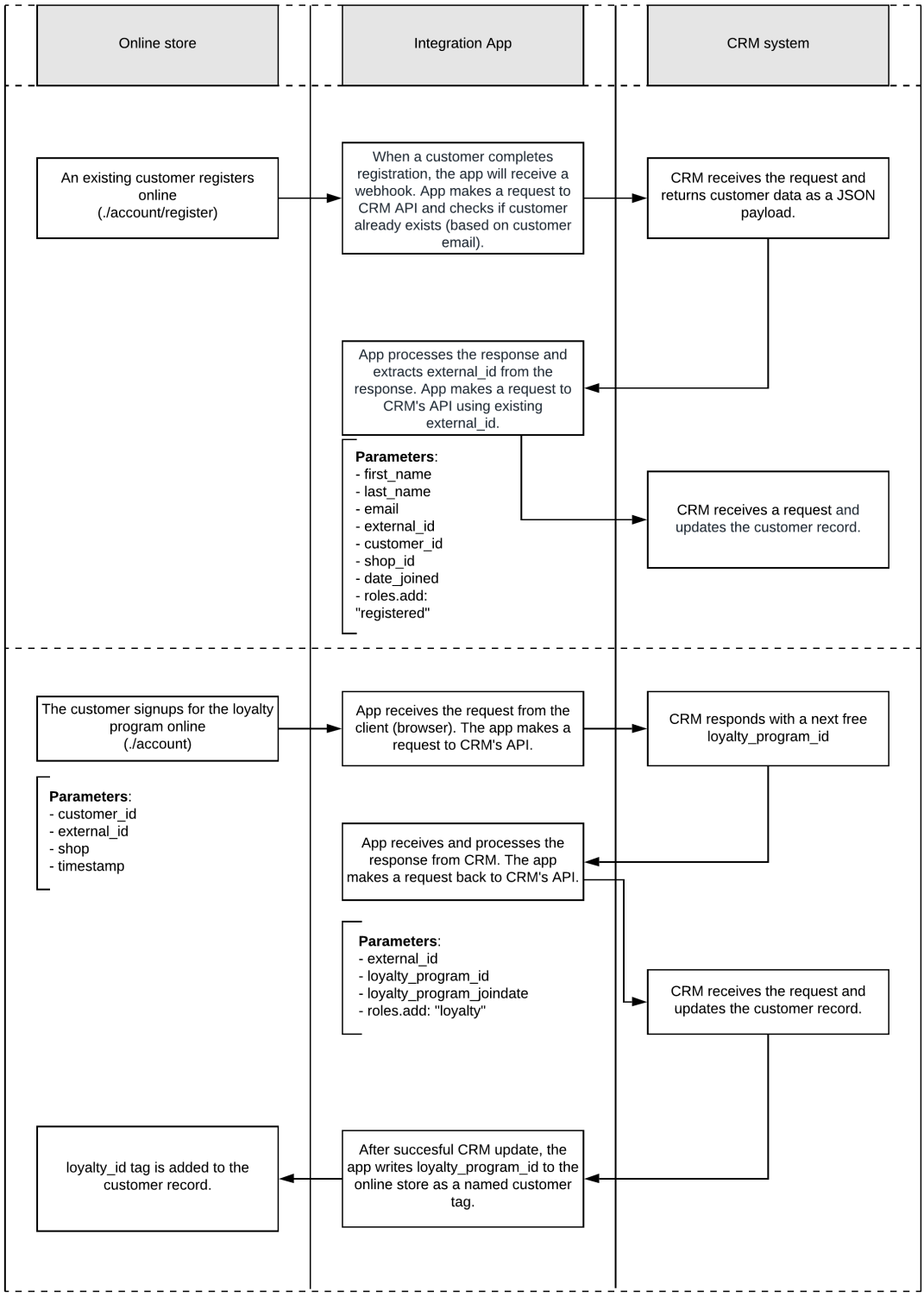


Figure 21. Flowchart for registration process and loyalty program signup

Let's take a closer look at the diagram above (Figure 21). As we can see, during the process you get the fifth identifier which is the loyalty card number (loyalty_program_id). It should also be noted that in the first CRM API call the identifier is your email address. In the very next call, the identifier is your external id. This is the happy case though, meaning that the CRM API returns one, and only one external id (no missing or duplicate data etc.). Moreover, when the integration application writes the loyalty card number to the online store's customer record (last row in the diagram), the identifier in the REST API request is your customer id. In total, four identifiers out of five are essentially involved in this single use case.

3.4.5 Friday 13 March 2020: Context Mapping

Today I try to wrap up my practical DDD experiment. On the systems-level, all the different customer identifiers must be mapped to each other. Without good architecture design and documentation, the team might get caught up in a conceptual confusion. It is also clear that developers cannot succeed without good input from domain experts. This is especially important with large and complex systems, like the one we have at hand.

As an example, the operating environment of a storage worker is very different from the one that a cashier or a marketing specialist has. The risks are greater the more poorly employees interact outside of their "silos". It is not unusual that in a situation like this no single business domain expert can communicate the system-level requirements to the development team. As an architect, you have to have several discussions with different departments and learn the grammar in every one of them. The output should be some kind of a "context map" that articulates the key concepts and their relationships. So, the last step of my DDD experiment is to delineate such a diagram. I focus on the integration between the online store and CRM system. As a result, I get a mapping as shown below (figure 22).

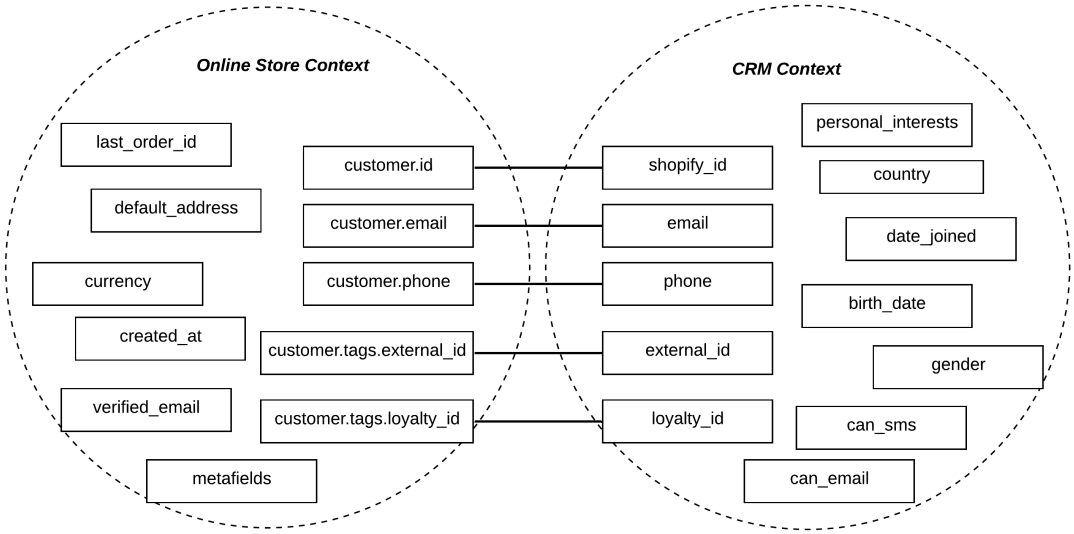


Figure 22. Context Mapping

This diagram (Figure 22) clearly brings out the relationships between the Bounded Contexts. It helps to communicate some of the system-level requirements. It also points out a few problems. As an example, different systems validate data in different ways, which can cause problems in this case. The ecommerce platform validates customer emails in two ways. Firstly, with valid form (so an @ symbol and a domain at the end basically). Secondly, it validates that the domain has a valid MX record. This last step verifies that the domain has a valid mail sever configured.

From the online store's perspective this is a good thing because invalid email addresses have no use (e.g. customers would not receive order confirmations and so forth). However, CRM's database might contain old customer records with invalid email addresses. This is not very problematic in the context of CRM if the customers have valid phone numbers and postal addresses. Marketing campaigns would still reach them. Nevertheless, these customer records cannot be migrated to the online store *at all*, because the data validation fails, and the REST API calls would end up in an error.

In fact, we have exactly the same problem with phone numbers too. According to the online store's API documentation (https://shopify.dev/docs/admin-api/rest/reference/customers/customer), this customer property can be set using

different formats, but each format must represent a number that can be dialed from anywhere in the world. For example, the following formats are all valid:

- 6135551212
- +16135551212
- (613)555-1212
- +1 613-555-1212

Unfortunately, the CRM system does not have a validation like this. Let's say that you work as a cashier during a Christmas peak. The brick-and-mortar is crowded with consumers, the line is long, and you try to keep up the pace. Next customer wants to join the loyalty program in store to get a discount from the book she is going to buy for a present. You just quickly create a new customer record to the CRM system via cash register. You ask for the name and email address, and that's all. You have to but *something* to the required phone field, so you just type "123456". New loyalty program member is created successfully, and the customer gets a discount. All clear – *in this context*. However, this customer record cannot be imported to the online store because the data validation fails, and the REST API calls end up in an error. From the online store's perspective, this loyal customer simply does not exist.

3.4.6 Weekly Analysis

This week I have studied domain-driven design (DDD) and its main concepts such as domains, subdomains, Bounded Contexts, and context maps. I have made a practical experiment and applied these concepts to a domain of customer management. Based on this experiment, I consider DDD as a way of learning more about a business domain and its problems. I feel it is a valuable approach if you need to model complex systems and communicate a design to various teams.

According to Nick Tune (2017), the co-author of "Patterns, Principles, and Practices of Domain-Driven Design", to make good strategic technical decisions you need to be immersed in the business context.

"You need to know what business goals you are working towards in order to decide how to optimise your tech strategy. Are you optimising for time-to-market or long-term stability? Is this piece of a work a good opportunity to train junior members? Is

now a good time to pay back technical debt? Only when you know business goals, can you make those kinds of decisions effectively." (Tune 2017.)

The author states that a technical strategy can be seen as a "pyramid of needs" (see Figure 23), where each layer builds upon the previous one. There are many ways for learning about the business. Firstly, the Business Model Canvas (for more information: https://en.wikipedia.org/wiki/Business_Model_Canvas) can help you understand the most valuable products or services the business provides, and the customers you should care about the most. Secondly, you can utilize the Product Strategy Canvas (for more information: https://melissaperri.com/blog/2016/07/14/what-is-good-product-strategy) for collaborating with business domain experts. Thirdly, you should gain a good understanding of team priorities, their architecture and their domain models. Tune recommends going and spending a few days working with the teams. Above all, this information should drive your architectural decisions. (Tune 2017.)

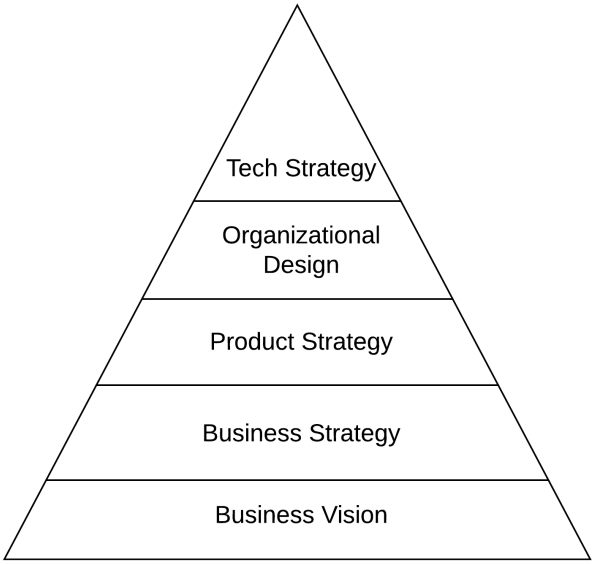


Figure 23. Strategic design (adapted from Tune 2017)

According to Tune (2017), experimenting with DDD and canvases should provide a solid foundation for strategic design. As an architect, the key is to ensure that every

decision you make is built on all the layers of the “tech strategy pyramid” (Figure 23). Quite a requirement in a hectic project environment like ours, I have to admit.

3.5 Week 5

3.5.1 Tuesday 17 March 2020: Support Process

We had a weekly meeting with the architect team today. We have been thinking of providing general instructions for contacting the architect team to get help. This is especially important now that we are all mainly working remotely due to the Coronavirus pandemic. We have identified three functions of the company that need our support on daily basis (see Figure 24). We get requests from sales (presales phase), project teams (implementation phase) and the support team (service phase). We also need to follow recent updates of the ecommerce platform (Shopify) on which we build on.

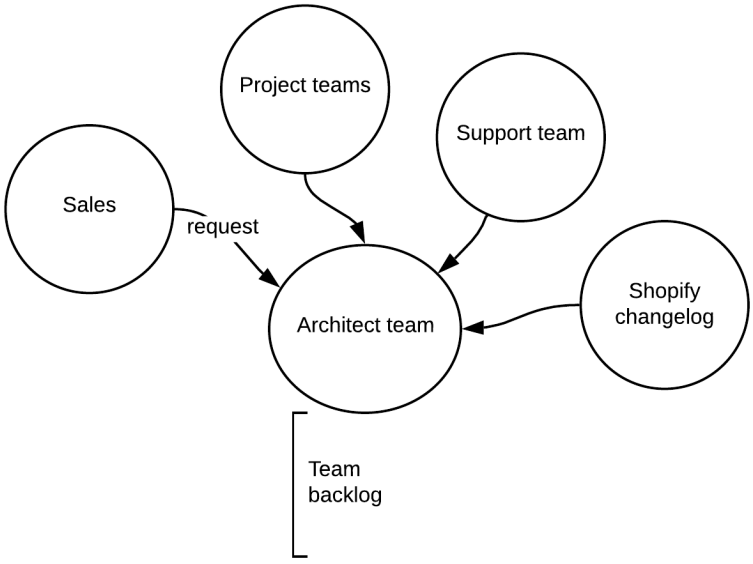


Figure 24. Support request flow

The main question here is that how the requests should be submitted to the architect team. We want to avoid interruptions through personal communication

channels and prefer transparency and cooperation. After the discussion we agreed on two good ways to get help from us.

Option A: *Use of a proper Slack channel.* This is a good option when the request is unstructured and unclear. You can start a discussion in dedicated Slack channels:

a) Ask in **#research** channel if it is something that we as a company should look into, as an example, if it has a potential effect on our customers or our business in general.
b) Ask in **#sales**-support channel if it is part of a presales work to a new customer.
c) Ask in a **customer specific channel** if it is something that is requested by an existing enterprise-level customer.

If it is possible, also communicate the importance of the request (e.g. the size of a sales case or the severity of the problem) and urgency (when a response is needed and why it is needed in that given time).

Option B: *Use of Teamwork Projects***.** You can create a task directly to Teamwork. In this case, the request goes to the backlog of the architect team, and it will be part of our weekly task prioritization and work sharing.

1. Create a task.
    a. Create a task in **"Research"** project if it is something that we as a company should look into, e.g. if it has a potential effect on our customers or business in general.
    b. Create a task in **"Sales support"** project if it is part of a presales work to a new customer.
    c. Create a task in a **customer specific project** if it is something that is requested by an existing customer. (If there is no project for the customer, just use "Sales support".)
2. Add the tag **"architect"** on the task. This is very important because tasks are filtered on the architect's Master Board by this very tag.
3. Put the task in the board column **"TODO".**
4. Finally, send a notification to **#sales-support** channel in Slack with the link to the task. By this you make sure that we as a team are notified about the new request.

I think this is a good start. We can share the first instructions via newsletter or Slack and get feedback from the employees. Additionally, we discussed internal Service Level Agreement (SLA). Should we have one? It makes sense to provide some kind of a service promise to the rest of the company. We decided to give a 1-day response time to any request pointed to the architect team. Depending on the request, the answer contains a solution, or at least an estimation of the resolution time.

3.5.2 Wednesday 18 March 2020: Architecture Styles

During the upcoming weeks I want to explore common software architecture styles and understand how they can help me on the way to become a better solution architect. According to Paul Clements and colleagues, architects can use *styles* as a starting point for their design. A style provides a generic solution approach to the problem at hand. However, it is just a guideline that needs to be refined by the architect and expressed in the end result – which is a view. (Clements et al. 2010, 35.)

The authors seem to be very explicit with this definition. As an example, they state that *"when we apply a style to the system, the result is a view"* (ibid., 29). Some pages later the same message is repeated in slightly different words: *"the emphasis here is on how to document a view that results from the use of a style"* (ibid., 49). Furthermore, each style they present in their book falls into one of these categories (ibid., 45):

1. *Module* styles help to think about a software as a set of implementation units.
2. *Component-and-connector* styles help to think about a software as a set of elements that have runtime behavior and interactions.
3. *Allocation* styles help to think about how a software relates to non-software resources in an execution environment.

As an architect, you should have an understanding of each of the three categories of architecture styles. In this respect, my personal learning map looks something like this (Figure 25):

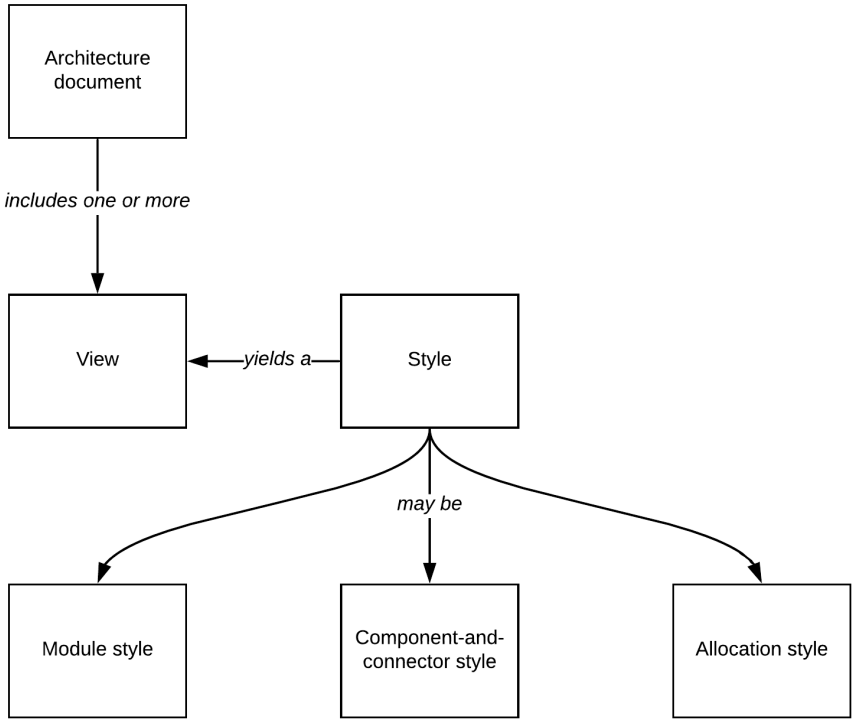


Figure 25. Architecture styles overview (adapted from the inside cover of Clements et al. 2010)

I am going to study each style and try out if they can guide me to better design decisions and architecture documents.

3.5.3 Thursday 19 March 2020: Module Styles

Today I am studying module views which can be used for education, stakeholder communication, and the basis for systems construction. Why is this important? According to Clements and others, it is unlikely that an architecture documentation can be complete without at least one module view. The authors define the term *module* as an implementation unit of software that provides a coherent set of responsibilities – such as its functionality and the data it maintains. (Clements et al. 2010, 55-56.)

Clements et al. introduce a half dozen of module styles in their book (2010, 65-122). I decide to choose a few and try them out. I start with the decomposition style.

"A decomposition view presents the responsibilities of a system in intellectually manageable pieces that are refined to convey more and more details. […] This style is an excellent learning and navigation tool for newcomers to the project and other people who do not necessarily have the whole functional structure of the system memorized." (Clements et al. 2010, 67.)

The authors state that almost all architects begin with this view. It is kind of a "divide-and-conquer" approach that helps to break a complex system into smaller sub-problems. The idea is that you can design solutions to the sub-problems, and finally combine these solutions to construct the system itself. I consider this as a good option especially in our context. In principle, we try to utilize the Shopify ecosystem as much as we can in our ecommerce projects. If the core functionalities are not enough, then we look for a solution in the Shopify App Store containing over three thousand apps to choose from. We have found that this speeds up time-to-market for our projects. Another option is to reuse self-developed modules from our previous projects. Moreover, if we need to build something totally new, we can design it so that it will help us later in other projects. (ibid., 65-66.)

Modules in the decomposition style are usually depicted as named boxes that contain other named boxes. So, how would a typical ecommerce system look like in a view like this?

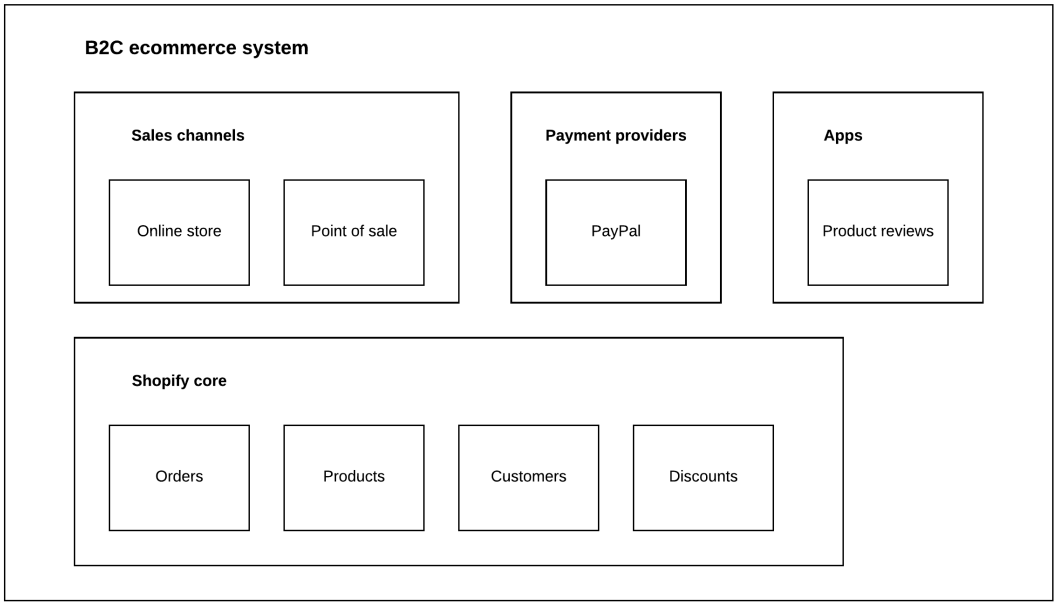


Figure 26. Decomposition view of an ecommerce system

I find this kind of a diagram (Figure 26) useful especially during the project planning phase. As an example, when you are scoping a project you might ask: do we need to make any data migrations? If yes, what kind of data? How many products or existing customers should be migrated to the new store? The diagram also makes it easier to communicate to stakeholders that an ecommerce store can have several third-party apps or payment providers. Additionally, every order is recorded to Shopify's database in spite of the sales channel. However, the diagram does not show all the dependencies between the modules or how they interact with each other.

It is interesting that you can go more into details by just adding a new decomposition view. Let's say that we are particularly interested in one of the sales channels – the online store. We can combine the previous view to this one (Figure 27):

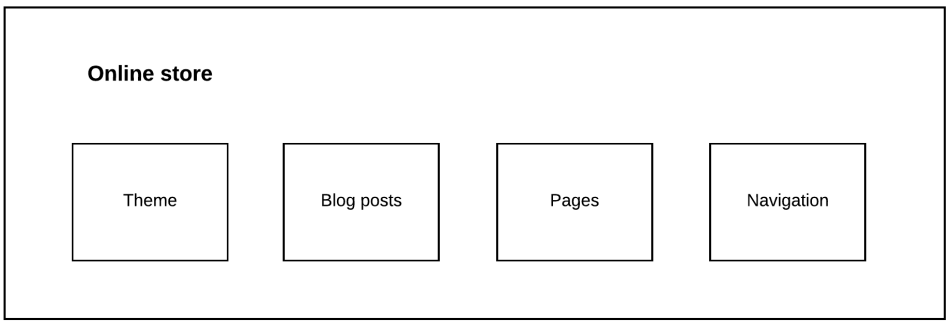


Figure 27. Decomposition view of an online store

And in the same way, if we want to communicate how a Shopify theme is structured, we can simply add yet another decomposition view to the document (Figure 28).

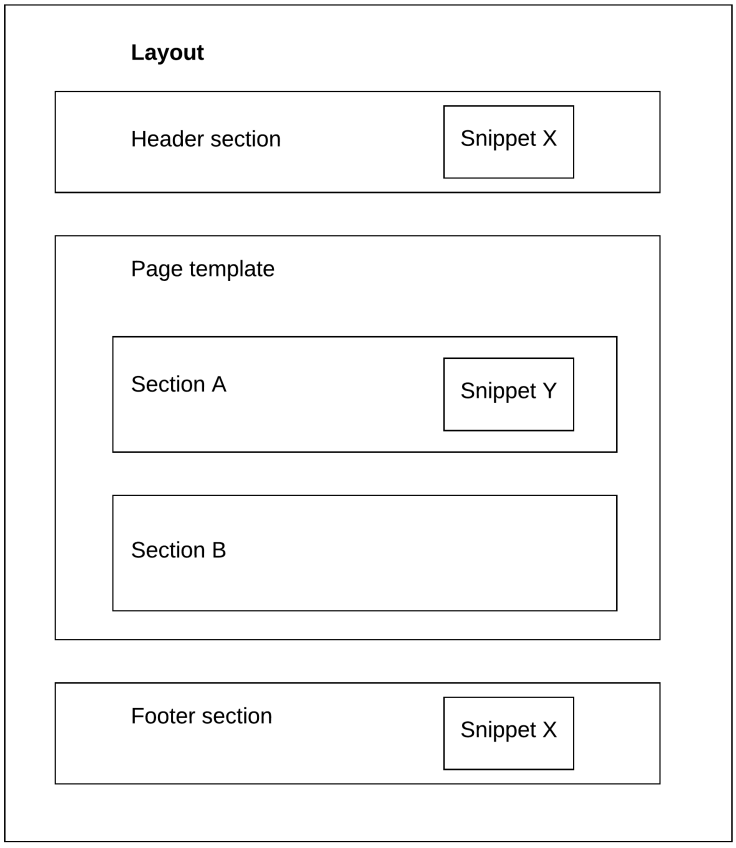


Figure 28. Decomposition view of a Shopify theme

As we can see, Shopify themes are made up of Liquid files, each serving their own unique purpose (Theme templates):

- Layout contains elements that are repeated on all page templates.
- Template is the actual content area, such as the content of a home page, collection page or product page.
- Sections are reusable modules of content that can be customized and re-ordered by users of the theme (e.g. slideshow).
- Snippet files are bits of code that can be referenced in templates of a theme (e.g. search bar).

3.5.4 Friday 20 March 2020: Module Styles

Today I am going to apply another module style which is called the data model. According to Clements and other, the output of data modelling describes the static information structure in terms of data entities and their relationships. Such a data model is often represented graphically in entity-relationship diagrams (ERDs). Any distinguishable object that contains information (to be stored or represented in the system) can be a data entity. (Clements et al. 2010, 109-111.)

The authors state that a general way to do data modelling is to start with a draft view. In an early stage, the documentation may contain key entities and important relationships. Over time, as design decisions are made, this high-level model is elaborated by the architect into a model that shows details of how the data is structured and stored. (ibid., 109-111.)

Why is this important? Firstly, the data model facilitates stakeholder communication during domain analysis and clarification of requirements. In our business context, relevant questions could be such as “what order data is needed for printing shipping labels for DHL Express packages” or “what product data is needed to create a Google Shopping campaign”. Secondly, the data model guides development teams in implementation of modules that access the data. Thirdly, it is a means to impact analysis of changes to the data model. This kind of modifications to an existing system can be time consuming and costly, as they may require changing the code of multiple applications that share the data. (ibid., 111-115.)

I am interested in seeing how this documenting process would look like in practice. I start with the *conceptual* data model that focuses on key entities and relationships in the given domain (Clements et al. 2010, 110). One recurring task for us architects is to design how to migrate orders from Shopify to on-premise ERPs. If the Shopify platform's order data structure is well documented, it helps to complete the customer specific architecture. So, I choose this topic for my experiment. I start by looking at an actual order JSON payload from my development store (Figure 29).


```
{
  "order": {
    "id": 2181376737376,
    "email": "jari.suni@woolman.io",
    "closed_at": null,
    "created_at": "2020-03-13T09:59:27-04:00",
    "updated_at": "2020-03-19T06:06:39-04:00",
    "number": 24,
    "note": null,
    "token": "0cbf825dc481c9b326a95d14f9e4b41e",
    "gateway": "bogus",
    "test": true,
    "total_price": "170.00",
    "subtotal_price": "170.00",
    "total_weight": 0,
    "total_tax": "32.90",
    "taxes_included": true,
    "currency": "EUR",
    "financial_status": "paid",
    "confirmed": true,
    "total_discounts": "0.00",
    "total_line_items_price": "170.00",
    "cart_token": "59825119ff93af0c9c5ab046eb808fa7",
    "buyer_accepts_marketing": true,
    "name": "#1024",
```


Figure 29. Order JSON payload

The payload starts with an order id and customer email, and is followed by some payment details etc. The actual content of the order can be seen from the next screenshot (Figure 30).

```
"line_items": [
  {
    "id": 4703319982176,
    "variant_id": 20237348208736,
    "title": "ADIDAS | SUPERSTAR 80S",
    "quantity": 1,
    "sku": "AD-01-white-5",
    "variant_title": "5 / white",
    "vendor": "ADIDAS",
    "fulfillment_service": "manual",
    "product_id": 2290764316768,
    "requires_shipping": true,
    "taxable": true,
    "gift_card": false,
    "name": "ADIDAS | SUPERSTAR 80S - 5 / white",
    "variant_inventory_management": "shopify",
    "properties": [],
    "product_exists": true,
    "fulfillable_quantity": 1,
    "grams": 0,
    "price": "170.00",
    "total_discount": "0.00",
    "fulfillment_status": null,
```


Figure 30. Order line item

From Shopify's perspective, the conceptual data model for order processing looks something like this (Figure 31):

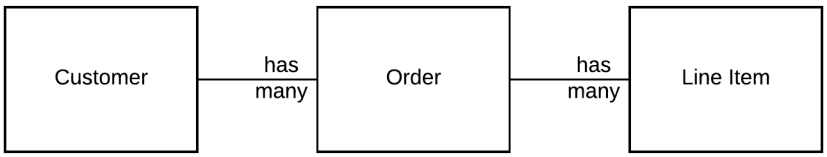


Figure 31. Conceptual data model (adapted from Clements et al. 2010, 110)

According to Clements and colleagues, the next step is to draw a *logical* data model that is evolved from the conceptual data model. As mentioned before, such a data model is often represented graphically in entity-relationship diagrams (ERDs). The authors state that one of the most popular ERD notations for relationships uses lines with special symbols at each end. These symbols typically include a dash (indicating one), a ring (indicating zero), and a crow's foot (indicating many). This fun named

Crow's foot ERD notation has a long history dating back to the 80s. So, what it would look like? (Clements et al. 2010, 116.)

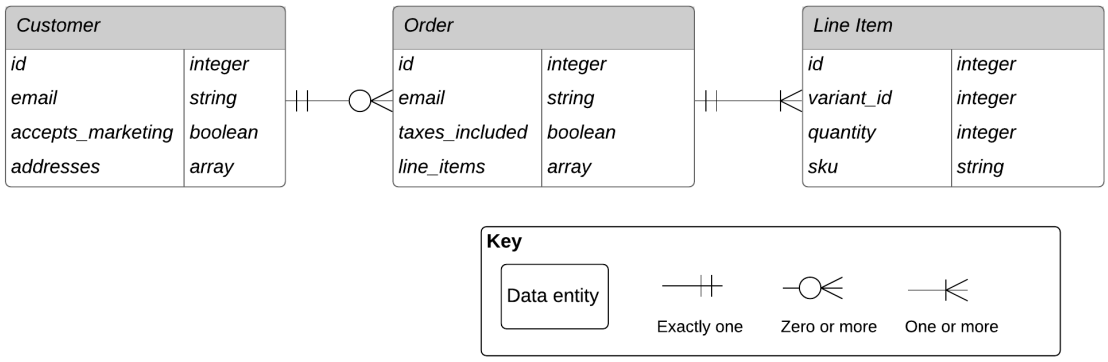


Figure 32. Logical data model (adapted from Clements et al. 2010, 110)

This simplified ERD (Figure 32) tells us the following:

- Each customer can have zero or more orders. As an example, it is possible to import customers to the system without any order related data. Additionally, if a new user signs up to a newsletter, a customer is created. Moreover, if you make multiple orders using the same email address, these orders are combined with the same customer record.
- Each order has exactly one customer.
- Each order has one or more "line items". As an example, you can order a t-shirt and jeans. In this case, the order has two line items.
- Each line item has exactly one order.

In fact, we could continue this experiment and connect line item's *variant_id* to Shopify's product data structure and so forth. However, let's stop here for now. It is easy to see what value this kind of a diagram adds to an architecture document. As an example, the ERD above helps to communicate how to manage fulfillments for orders. Let's say that you have bought a t-shirt and jeans. T-shirts are in stock, but jeans are still on the way from the supplier to the warehouse. In this case, it is possible to make a partial fulfillment and send a t-shirt to you right away. This function can be done via Shopify's Fulfillment API using a HTTP request:

POST /admin/api/2020-01/orders/#{order_id}/fulfillments.json

```
{
  "fulfillment": {
    "location_id": 905684977,
    "tracking_number": "1234567",
    "tracking_url": "http://www.packagetrackr.com/1234567",
    "tracking_company": "Some Package Tracking Company",
    "line_items": [
      {
        "id": 466157049
      }
    ],
    "notify_customer": true
  }
}
```

In this example, an order is partially fulfilled by specifying a line item id in the request. As a customer, you will get a shipping confirmation with a tracking number for this shipment (for more information: https://shopify.dev/docs/admin-api/rest/reference/shipping-and-fulfillment/fulfillment).

Let’s continue the data modelling experiment a bit further and link order line items to Shopify’s product data structure. This can be visualized like this (Figure 33):

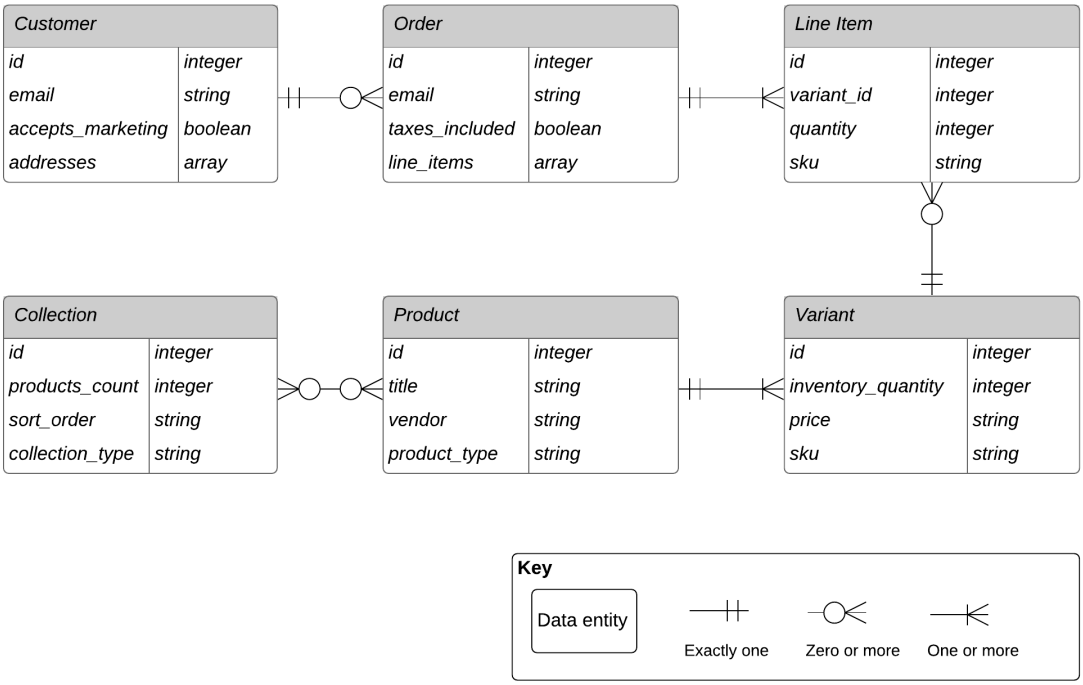


Figure 33. Entity-relationship diagram

As can be seen in the simplified ERD (Figure 33), each order line item has exactly one product variant. The other way around, each variant can be purchased zero or more times. Furthermore, every product has at least one variant. In Shopify, you can create up to 100 variants for a product. Additionally, it is possible to group products into collections to make it easier for customers to find them by category. Each collection can have zero or more products, and vice versa. A product can belong to more than just one collection. In other words, a product can be in several collections such as “shoes”, “men”, and “on sale”.

3.5.5 Weekly Analysis

Our working routines changed a lot this week due to the Coronavirus pandemic. Basically, every one of us is working remotely for now. As a team of architects, we needed to re-evaluate our own operating model too. We started the week by creating a better “demand control” and set up a formal process of communication to the architect team. We believe that this increases the overall transparency and cooperation. I also started a new routine this week. I established a support hour: same time every day I am available online for any questions regarding the client

work. This helps me to plan the workday in advance and reserve also enough time for unpredictable support work.

On a professional level, I am more confident about the importance of the task at hand: developing a practice for documenting design decisions and software architecture so that the team can easily use the output (whatever it is) and build a working system of it. In a way, I try to make myself unnecessary. By this I mean that ideally architecture documentation should be self-sufficient. Of course, this creates a lot of pressure on its technical quality and clarity. On a personal level, I find the learning of practical documenting skills rewarding. I have been studying module styles this week, mainly the decomposition style and data model style. At the moment, my learning path looks like this (Figure 34):

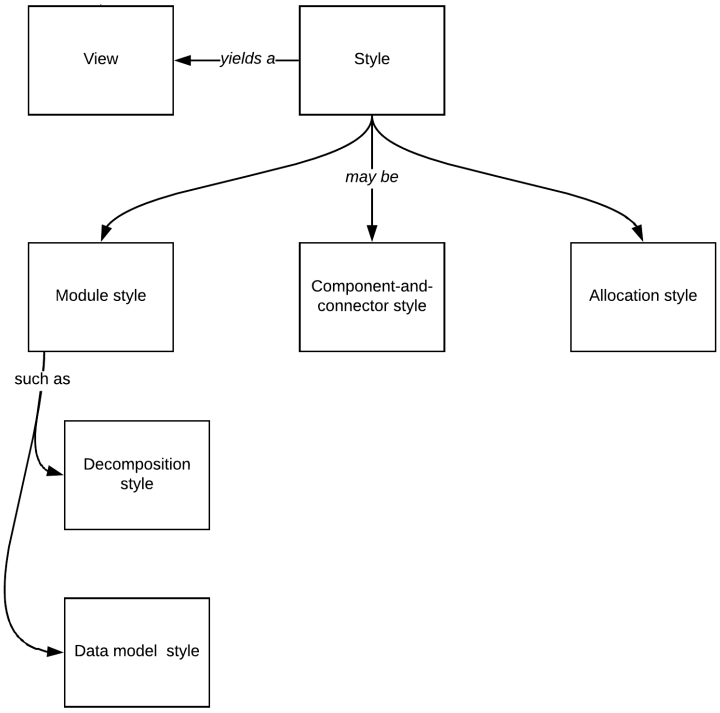


Figure 34. Learning path to architecture styles (adapted from the inside cover of Clements et al. 2010)

I am not only interested in the technical notation of each style. It should be noted that a style also brings on a practical process for building a view. I like both the “divide-and-conquer” approach in the decomposition style, and the “top-down”

design methodology (from conceptual to logical) in the data model style. Because of this, it will be exciting to see what the component-and-connector style and the allocation style have to offer.

3.6 Week 6

3.6.1 Tuesday 24 March 2020: Component-and-Connector Styles

Today I continue to study architecture styles and move on from *module* styles to *component-and-connector* (C&C) styles. What is the main difference between these styles? When module views describe how the system is structured, C&C views are commonly used to show how the system works. In this respect, they are an important addition to the architecture documentation while specifying the structure and behavior of runtime elements. These elements can be such as processes, clients, servers, and data stores, accompanied with protocols, information flows and database access. (Clements et al. 2010, 123, 136.)

According to Clements and colleagues (2010, 155), the space of C&C styles is rather large. To make sense of this diversity, I like to start with some broad categories of commonly used C&C styles before going into the details like notation and so forth. So, what are these general categories? The authors state that C&C styles can be distinguished largely by their underlaying computational model (ibid., 156).

- *Data flow styles* embody a computational model in which components act as data transformers. Connectors transmit data from the outputs (of one component) to the inputs (of another component). This architecture style is driven by the flow of data through the system. (ibid., 156-157.)
- *Call-return styles* embody a computational model in which components provide services and capabilities that may be invoked by other components. Connectors convey requests (from one component to another) and are also responsible for returning any results for the requests. (ibid., 161.)
- *Event-based styles* allow components communicate to each other through asynchronous messages. In such systems, events trigger behavior in other components. Sometimes connectors are point-to-point, but sometimes an event is sent to multiple components. (ibid., 172-173.)
- *Repository styles* describe system components which typically retain large collections of persistent data. Other components read and write data to these shared repositories. In many cases, access to a repository is mediated by software that provides a call-return interface for data retrieval and manipulation. (ibid., 178.)

In general, such architecture styles describe how execution and data *flow* through systems. Moreover, many C&C views involve components that run as concurrent processes. The authors emphasize that in these cases it is also important to document how these processes are scheduled and prioritized among each other. (Clements et al. 2010, 185-186.)

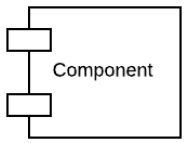
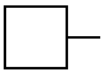
3.6.2 Wednesday 25 March 2020: Component-and-Connector Styles

Today I am going to study what are the preferred notations and processes for building a component-and-connector view. As usual, box-and-line drawings are an obvious option for representing C&C views. However, in this case Clements and colleagues are more accurate. The authors state that you should pay special attention to the connectors: *"A common source of ambiguity in most existing architecture documents is the meaning of connectors, especially with ones that use arrows as their visual symbol."* (Clements et al. 2010, 139.)

According to Clements and others, you should rather use "semiformal notations" like UML components because they are a good semantic match to C&C components such as interfaces and behavioral descriptions. As an example, UML *ports* are a good match to C&C ports. Moreover, UML provides a *lollipop/socket* notation for showing interfaces attached to ports. These interfaces can then be further elaborated by supplying additional information (methods, attributes etc.) in the architecture document. (ibid., 139-141.)

However, I am not familiar with ports, lollipops or sockets. Since I am not an expert in UML, I have two options. Either step back and use informal notation for my C&C views or learn the basics of UML. I decide to follow best practices and learn the UML notation needed in C&C views. So, what are the basic "shapes" and their meaning? For me, a good place to start is the documentation of the diagramming software we are using at work. According to Lucidchart's documentation (Component Diagram Tutorial), the following symbols are commonly used for building component diagrams (Table 4):

Table 4. Common symbols for component diagrams (adapted from Component Diagram Tutorial)

Symbol	Name	Description
	Component symbol	An entity required to execute a function. A component provides and consumes behavior through interfaces, as well as through other components.
	Provided interfaces (Lollipop symbol)	Represent the interfaces where a component produces information used by the required interface of another component.
	Required interfaces (Socket symbol)	Represents the interfaces where a component requires information in order to perform its proper function.
	Port symbol	Specifies a separate interaction point between the component and the environment.

Next steps are to find a good use case and make a pertinent diagram. In fact, I have been looking for an expressive way to communicate the logic of the proxy app we have developed for our client. It is kind of a customer specific add on that is needed for retrieving data (like earned loyalty points) from standalone CRM system to the online store's My Account page and displaying it for the end-user. As I see it, the best way to describe this is to use *client-server style* which is part of the general category of call-return styles. As the name implies, in this case the component types are *clients* and *servers* (Clements et al. 2010, 162):

- Servers have ports that describe the services they require.
- Clients have ports that describe the services they require.
- Servers may also act as clients by requesting services from other servers.

According to Lucidchart's documentation (Component Diagram Tutorial), I also need to include the component type (e.g. "client" or "server") inside the double angle brackets, and the component symbol. This is important because a rectangle with just

a name inside of it is reserved for class elements in UML. All this is new to me, but let's try this out. My first draft looks like this (Figure 35).

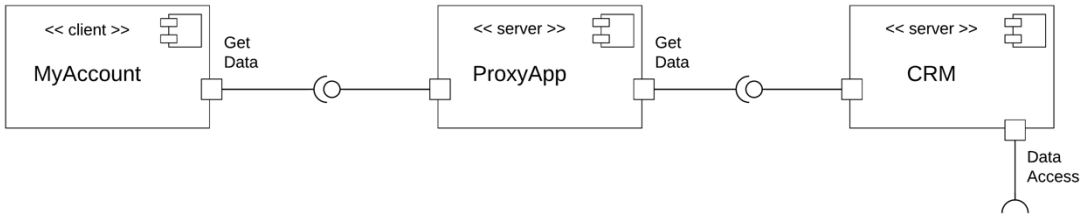


Figure 35. A client-server view

So, when a user navigates to the online store and logs in, the client (web browser) makes a request to the endpoint provided by the application proxy. The request is forwarded through the proxy to the application programming interface of CRM.

3.6.3 Thursday 26 March 2020: Component-and-Connector Styles

Today I continue my practical experiment with C&C styles and UML notation. As mentioned earlier, I have to pay attention to connectors and interfaces. To be honest, my current drawing is rather undetailed in this sense. I want to provide more information in my view but how it should be presented? To my surprise, I cannot find good examples anywhere. I start to believe that there must be a reason for this. After studying for quite some time, I find an advice of Clements and others.

"The component-and-connector types instantiated in a particular C&C view should be explained by referring to the appropriate style guide that enumerates and defines them. […] The definition of a component or connector type should characterize the number and type of interfaces […] that instances of the type can have." (Clements et al. 2010, 131.)

After understanding this the pieces start to fall into place. As an architect, I should document the component types in a *supporting* documentation – not in the view itself. The authors state that a *type* is an incompletely defined component or connector. An *instance*, described in a view, is the result of completing that

definition. Moreover, each instance must conform to its type in terms of behavior, interfaces, properties and so forth. In this sense, all instances of a given type are more or less identical to each other. (Clements et al. 2010, 129.)

For my practical experiment, this means unnecessary work. I am documenting *a view*, not pursuing descriptive completeness of all elements and relations in the system. However, if you are documenting a large system, this approach makes sense indeed. According to Clements and others (2010, 130), it is useful to identify elements with common logic and locate this information in a *type definition* (as opposed to replicating it across each instance in every single view). This sort of systematic approach makes it easier to understand and communicate the overall system.

Now back to work. Despite this advice of Clements and colleagues, I try to supply additional information directly in my architecture view. Eventually, I find some hints on how to do it from the appendix of the book (Clements et al. 2010, 438-443). I decide to add a few notes and connect them to the APIs.

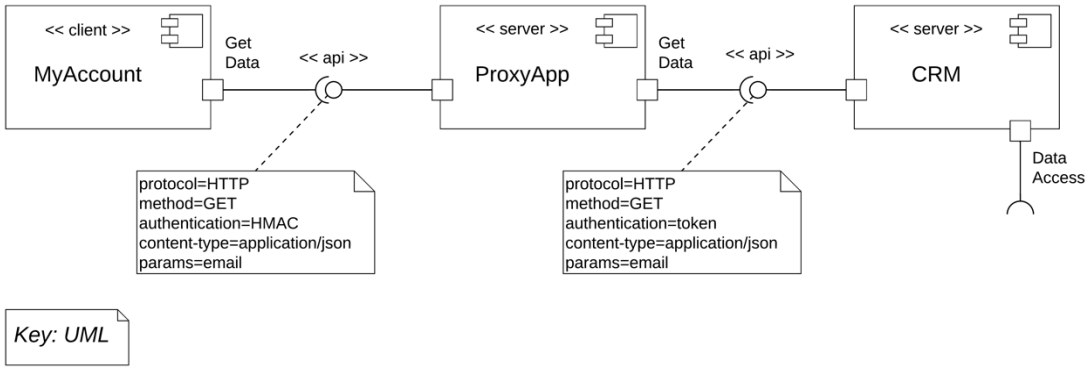


Figure 36. A client-server view with API descriptions (adapted from Clements et al. 2010, 440-441)

As can be seen (Figure 36), I have only two interfaces in a single view and I am already repeating myself. The only information that changes between the notes is the authentication. In the first API, a HMAC signature is expected to be provided in the HTTP request header. In the second API, the method of authentication is a user-specific access token (passed in the HTTP request header). With that in mind, all the

other information could be transferred from the notes to a supporting documentation and tied together with the <<api>> stereotype. In the same way, ProxyApp and CRM are subtypes of the <<server>> stereotype. Furthermore, these C&C types could be mapped to *modules* used in other views of architecture documentation.

3.6.4 Friday 27 March 2020: Behavior Diagrams

Today I have been reflecting on my experiences on documenting architecture views using the module and C&C styles. I am pretty happy with what I have learned so far. Nevertheless, one thing left me bothering. I would like to document also the return messages sent by the applications that receive and process requests. For example, if the HMAC signature is invalid, the logic that follows should be different from the happy case. I would like to communicate the assumed scenarios identified at the design phase. I am interested in a view that is expressive is this particular sense. How to build such a view? Which would be the desired notation and so forth?

I believe that the go-to solution is some kind of a behavior diagram. According to Clements and others, behavior diagrams complement the structure diagrams found in module and C&C views. As an example, a *sequence diagram* can describe the behavior of the modules when executing a specific scenario of the system. The participants in a sequence diagram are called UML objects. Moreover, these participants may be UML component instances from a C&C view. (Clements et al. 2010, 449-451.)

I want to continue my practical experiment and build a UML sequence diagram for my use case. However, I am not familiar with the notation needed so I need to look for documentation first. According to Lucidchart's tutorial, sequence diagrams are a popular modeling solution in UML because they specifically focus on *lifelines* of objects, and the messages exchanged between them to perform a function before the lifeline ends. The tutorial introduces the basic symbols and components used in sequence diagrams. Especially the *alternative symbol* draws my attention. In short, it symbolizes a choice between two or more message sequences. This is exactly the frame I need to include in my view to be able to show a switch-case construct. (UML Sequence Diagram Tutorial.)

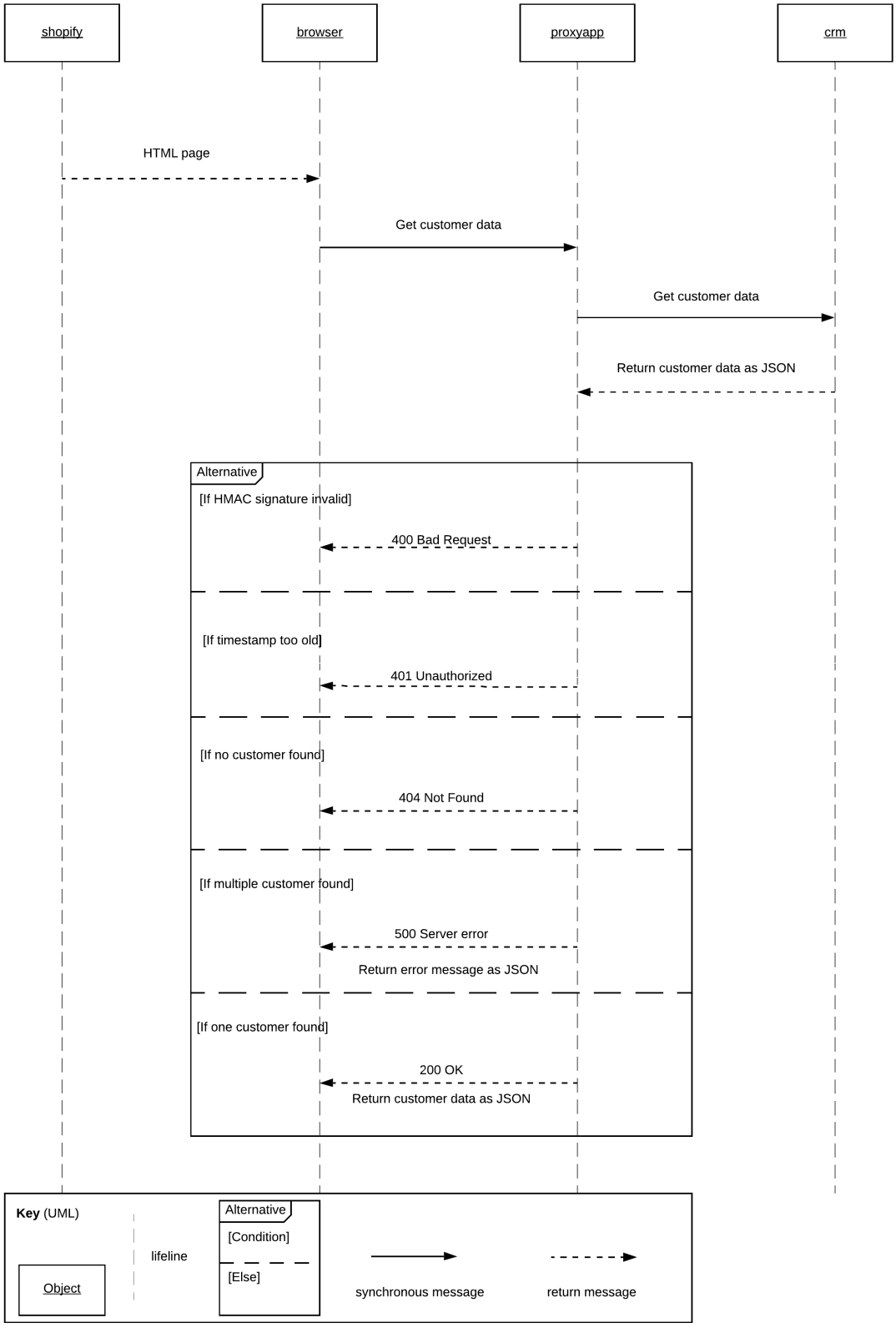


Figure 37. Detailed sequence diagram

As can be seen in the resulting diagram above (Figure 37), the different scenarios identified in the design phase are now communicated in the view for both backend

developers and frontend developers. For example, any error messages displayed to end-users can be designed according to the HTTP response status codes and payloads. Additionally, the view can be used to evaluate whether the design is sufficient in this sense.

3.6.5 Weekly Analysis

This week I have studied so called component-and-connector styles. I documented two application programming interfaces in a client-server view, definitely a good addition to the solution architect's toolbox. Finally, I also had to look more carefully to the Unified Modeling Language (UML) notation. I consider this to be perhaps the best issue of the week.

I also had an aha moment. In some sense, documenting a system is not very far from building it. Same principles, like Don't Repeat Yourself (DRY), applies to both architects and programmers. With a proper abstraction, you can produce much cleaner architecture (or code). I also understand the need to define *stereotypes* when documenting a large system with multiple views. They are like components which can be reused from another project or application.

In addition, I had to look for a way to document behavior, not just the structure. I found the idea of using a "timeline" (or a "lifeline" like they call it in UML) very rewarding. According to Felix Bachmann and colleagues, a view can have an associated description that documents such a behavior of the elements:

"Without taking into account how the elements behave when connected to each other, there can be no assurance that the system will work as intended. Achieving such assurances before the system has been fully implemented is a major goal of paying attention to its architecture. Element behavior, therefore, is an essential part of architecture and therefore of architecture documentation." (Bachmann, F., Bass, L., Clements, P., Garlan, D., Ivers, J., Little, R., Nord, R. & Stafford, J. 2002, 3.)

3.7 Week 7

3.7.1 Monday 30 March 2020: Allocation views

This week I would like to complete my study tour to architecture styles. Past two weeks I have been studying the module styles and C&C styles. In addition to these, Clements and colleagues also introduce the *allocation view* category, which is used to document non-software structures such as organizational environment and show how the architecture design is mapped to this environment (Clements et al. 2010, 189).

In this case, the basic idea is to use *the work assignment style* to allocate modules (of the module style) to individuals (of a team) who are responsible for the realization of the system (ibid., 202). My first impression is some kind of a surprise. Is it really architect's responsibility to assign work for developers? For better or worse, the authors seem to be very serious about it. They state that it is through the mapping between the software architecture and the team structure that project management activities can proceed in the first place (ibid., 189). In other words, architecture documentation should communicate to the organization also the skill set and effort that is needed for building a system from it. A rather startling thought, isn't it?

Furthermore, it is not just the developers who are building the system. Somebody needs to test and validate it too. Even if a module is purchased as a commercial "off-the-shelf" product, it must be installed and configured. Someone still has to be responsible for the module and "speak for it" during the implementation, as the authors point out. All these tasks have a place in a work assignment view. (ibid., 202.)

3.7.2 Tuesday 31 March 2020: Work Assignment Style

Today I am going to apply the work assignment style to our business context. I am interested in seeing what such a view looks like and what value it can possibly add. According to Clements and others, the work assignment style helps with planning the resource allocations and explaining the structure of a project. Moreover, this style can serve as the basis for project estimates like budget and schedule. (Clements et al. 2010, 203.)

This time the authors give a free hand for drawing such a view. You can use informal notations or just create a table, whatever you think is the best. For example, work assignments could be the modules from a decomposition view, or the software associated with tasks or processes in a system. The idea is to use the view with a project manager for dividing the project work into manageable chunks. (Clements et al. 2010, 203-205.)

We have started an ecommerce project for a jewelry brand. The project is extensive with many different areas. I consider it as a good example for this practical experiment. I decide to proceed with a tabular notation. We usually like to break down a project to task lists. Every task list has a detailed set of tasks and sub-tasks. To my understanding, the primary presentation for a work assignment view should be kept at an adequately general level. My first draft looks like this (Table 5).

Table 5. Work assignments for a project

Project area	**Task list**	**Team member**
Integration application	Order processing	backend developer
	Order statuses	backend developer
	Inventory levels	backend developer
Theme development	Homepage sections	frontend developer
	Collection page filters	frontend developer
	Product page components	frontend developer
Shopify apps	Email marketing	growth hacker
	Loyalty program	solution architect
	Store locator	solution architect
Data migrations	Product import	specialist
	Customer import	specialist
	Gift cards	specialist
Additional sales channels	Point-of-sale	solution architect
	Wholesale	solution architect

The left two columns (in Table 5) echo a typical ecommerce system's module decomposition structure (see Figures 26-28). The right column (in Table 5) describes the organizational units and skill sets that are likely needed in a project like this. Two thoughts come to mind when looking at the table. Firstly, this kind of a view could help to put together the right kind of a project team and make necessary work reservations. Secondly, such a view could come in handy when outlining the workload of the project and thus also the value of the bid. Perhaps the work assignment view should be created already at the presales stage.

3.7.3 Friday 3 April 2020: Work Assignment Style

Today I have been reflecting on the tabular notation I used for documenting the initial work assignments for an ecommerce project. I think it is missing one important viewpoint. We are doing projects for our clients. The success of a project requires that the client is closely involved in both definition and validation. In other words, many important work steps actually belong to the customer. However, the client shines with her absence in the work assignment view. Is there any good way for presenting this kind of an information? I am interested to see whether the UML has something to offer in this regard.

According to Clements and others, UML does not have a diagram type that is intended to show work assignments. Nevertheless, this does not mean that you should not use UML for such a view. In fact, the authors state that it is possible to take advantage of a package diagram with UML symbols like *actors* and *packages*. So, how it would look like? (Clements et al. 2010, 446-447.)

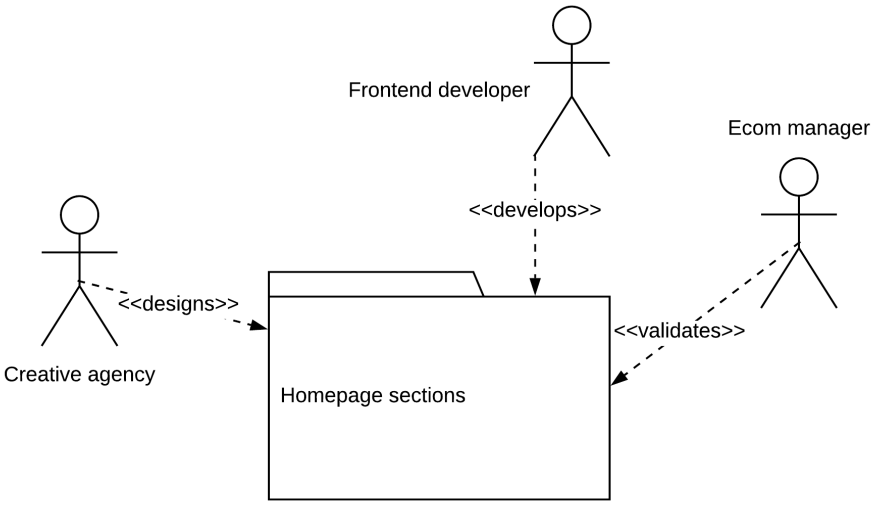


Figure 38. UML notation for a work assignment (adapted from Clements et al. 2010, 447)

As we can see (Figure 38), at least three parties are involved in the development of this module. I think that this kind of an information is valuable for our developer. It is clearly communicated that an external partner is strongly involved in the designing phase and affects the appearance of the ecommerce store. It is also defined who accepts the results of the development work.

3.7.4 Weekly Analysis

This week I have finally completed my study tour to architecture styles. I have focused on documenting non-software structures such as organizational units in project environment. I made a few practical experiments with the work assignment style. I noticed how this can help with planning the work and explaining the structure of a project. This made me think a bit more closely about the relationship between a project manager and an architect. It seems that the relationship is closer than I had let myself understand.

According to Malveau and Mowbray (2004, 167), a number of project management activities require the support of the architect. One of the main tasks is to establish the project technical vision. However, managing responsibilities do not end there. The authors state (ibid., 167) that as an architect you have to take care of these tasks as well:

- create technical work plans
- engage in team-building activities
- maintain the documentation of the technical strategy throughout the project's lifecycle
- identify project's technical challenges, risks and progress
- ensure long-term customer satisfaction

4 Discussion

"Small changes can produce big results–but the areas of highest leverage are often the least obvious." (Senge 2006, 63.)

This chapter answers the research questions presented in chapter 1.2. In addition, a dialogue between the results and the theoretical framework is developed as Kananen (2015, 38) recommends. In other words, I relate my own experiences and findings to the general body of knowledge on documenting software architecture. Topics for further research are presented at the end of this chapter, as well as a brief reflection on the pros and cons of the chosen research method.

4.1 Answers to Research Questions

The first research question was: *What are the best architecture styles and views for documenting ecommerce systems?*

When I started my research, my goal was to find the best possible architecture style and notation for documenting ecommerce systems. In retrospect, it is easy to see how naive that goal was. In fact, the greatest lesson is that an all-encompassing view is not even worth pursuing. Instead, the problem needs to be systematically viewed from different perspectives using different approaches. At this point I really want to emphasize the word *systematically*. One of the most important findings is that each architecture style brings on a practical process for building a view. As an example, the "divide-and-conquer" approach in the decomposition style and the "top-down" methodology in the data model style are both useful in their own way.

Any business can be screened and decomposed into logical processes. Moreover, each solution is relative; the pros and cons depend on the viewer. Yet this does not mean that every solution is just as good. The architect must be able to see beyond

the ambiguous and find the solution that best serves the whole. This can be done step by step, by taking the position of a particular stakeholder at time and sketching a view after another, as if you were turning a sculpture in your hand and looking at it from different angles. As an architect, you simply have to practice and master several different styles. There is no shortcut.

The second research question was: *How should architecture document be produced effectively so that the team can learn from it and build a working ecommerce system from it?*

As an architect, I always try to draw a diagram that is worth a thousand words. However, it is not uncommon that I have to use a thousand words to explain the diagram before anyone understands what I draw. This is not very effective for either party. One of the purposes of this study was to find solutions to this particular problem. Fortunately the effort has not been wasted.

One important insight is to avoid ambiguity by explaining your notation. The best way to do this is to include a key in diagrams. The key explains the meaning of every symbol used. A simple practice like this instantly increases clarity and, in my experience, also harmonizes diagramming practices within the team.

Another important practice is to review documentation for fitness of purpose. In other words, architecture document should be reviewed by the stakeholders for which it is written. This practice is valuable because it helps to spot any obscurity. For an architect, this means understanding the documentation as a process, not an end result. The documenting process follows the Plan-Do-Check-Adjust learning cycle:

- Find out what stakeholders need.
- Record design decisions.
- Check the resulting documentation.
- Package the information in a useful form.

4.2 Theoretical Implications

This subsection presents how architecture styles can be classified according to the information needs associated with them. This also helps in positioning new styles in relation to the findings of this study. The theory is reflected on the research results and my own experiences.

One important dimension of architectural documentation is resource management. As Paul Clements and colleagues nicely state, understanding *which views* to produce at *what time* and with *how much detail* can be reached only in the context of a concrete project. This is easy to agree with. To choose the appropriate set of views, you must identify the stakeholders that will depend on the documentation, and you must also understand each stakeholder's informational needs. (Clements et al. 2010, 316.)

Typical stakeholders in our business context are the project team (project manager, designer, frontend developer, backend developer), the project customer, the support team (as maintainers) and, of course, end users. The information needs of these stakeholders can be viewed in relation to different architectural styles. For example, to create a schedule, the project manager needs information about the modules to be implemented, with some information about their complexity and dependencies. Probably the project manager is also interested in work assignments and any organization-to-organization interfaces (for which the work of different teams needs to be aligned). However, this person does not so much care about the technical specification of any particular element or interface (beyond having the task done). (ibid., 316.)

The project customer is the one who pays for the development. This stakeholder (here simplified into a single entity) is interested in cost and progress as well as whether the resulting system will meet the business requirements. The customer also wants to know how the system under development will interoperate with other on-premise systems in that given business environment. With this experience it can be said that the perspective of the project manager and project customer are surprisingly similar. (ibid., 321.)

However, the information needs change substantially when the system and its architecture are viewed from the perspective of the developer or maintainer. The best starting point for both is the general idea behind the system. Then the details are highlighted. For example, the developer is interested in what the data model of the assigned element is, what the interfaces associated with it are, and if there are any code assets he or she can make use of. (ibid., 319.)

Maintainers will want to see the same information as developers; however, their needs are more difficult to predict in general. This is because future development needs can apply to *any* part of the system. In a good case a *decomposition view* allows them to pinpoint the location where the bug fix needs to be carried out or perhaps they are looking for a *uses view* to help build an impact analysis of the planned change.

According to Clements and others, maintainers will also want to see supporting documentation and understand the architect's original thinking. This can save them plenty of time by seeing already discarded design alternatives. (Clements et al. 2010, 320.)

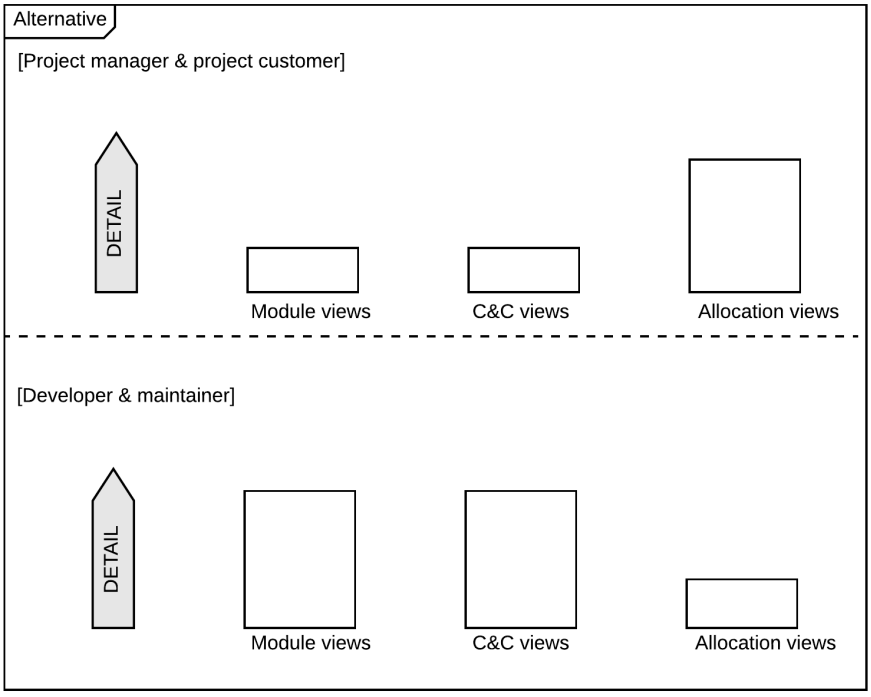


Figure 39. Stakeholders and their documentation needs (adapted from Clements et al. 2010, 317-322)

As we can see in the simplified diagram above (Figure 39), each architecture style has a different value depending on the viewer's context. In this sense, it is reasonable to include at least one view of each style to the resulting documentation. While the managers are usually interested in an overview information of the software to be

able to create a project plan, schedule, budget and work assignments, the developers' main interest is mainly in the software itself. As an architect, you cannot downplay the latter. As it is often seen, the devil is in the details. You can try to tackle this problem during a project by prioritizing the release of views to serve the most important project needs first. For the less critical business processes, the views can be left to a later date.

4.3 Practical Implications

At professional level, studying different architecture styles and applying them to practical work has been the best development for me. However, it should be emphasized that great strides have been made in the development of documenting processes as well. I will now list a few positive changes that we – as an architect team – brought about during the thesis writing process. I believe that these operation modes can be used successfully in any company or team.

We have set up a weekly ritual in which we work on our common task queue. We call it "architect backlog refinment". We use a dedicated Kanban board to visualize the work. The main objectives for the meeting are:

- refine enough backlog items so the architect team can be productive in the next sprint (week)
- establish shared vision and understanding about what is important
- identify early major goals, and plan longer term if possible

We have agreed on "definition of done". Each task must have at least a link to the documentation created during performing the task. Conversely, the task is not completed until it is also documented. A small thing that has had a big impact on the whole company.

In addition, we have established general instructions for contacting the architect team to get help. This has already reduced interruptions through personal communication channels and increased transparency and cooperation within the team. We also decided to give a 1-day response time to any request pointed to the architect team. Depending on the request, the answer contains a documented solution, or at least an estimation of the resolution time.

Once a month or at least every other month we get together and focus on the ways we do architecture work. We call these gatherings “Woolman Architects’ Day”. We have found that this is a good way to share lessons learned, harmonize working, and decide on new development targets.

In addition to this, we have decided what tools we will use for documentation. The main tools are:

- Teamwork Spaces for planning, specification and sharing knowledge both internally and externally. Main platform for architecture documentation. We call it our own wiki.
- Lucidchart – virtual workspace and our main tool for diagramming. These documents will be embedded to Teamwork Spaces and enriched with supporting documentation whenever needed.

During the writing process I have come to the conclusion that we should have document templates available for project specific documentation. The idea is that ready-made templates speed up and facilitate the documentation process. Once the templates become familiar to both architects and stakeholders, their structure helps to ensure that the design details are captured efficiently and understood correctly. In this case, the wheel does not need to be reinvented. Our first *view template* conforms to the standard introduced by Clements et al. (2010, 337-341), and it consists of two sections: *primary presentation* and *supporting documentation*. The template (see Figure 40) is saved to our wiki (Teamwork Spaces) and it is available for the architect team.

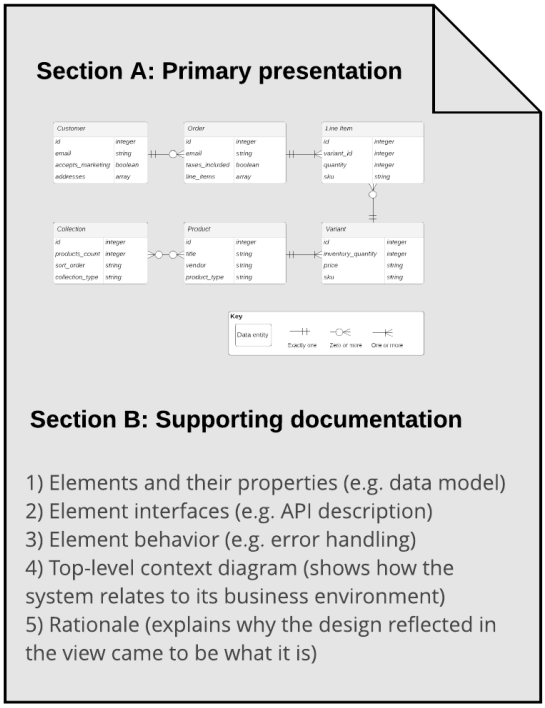


Figure 40. Template for a view (adapted from Clements et al. 2010, 338)

The primary presentation is most often graphical, e.g. a flowchart. Sometimes the primary presentation can be textual, such as a data mapping table. Regardless of the presentation style, its role is to present the most important information in the view. If needed, this may feature more than just one diagram. The second section is for supporting documentation. This section should reflect the architect's original thinking (e.g. discarded design alternatives). This information can save plenty of time later on. (Clements et al. 2010, 338-341.)

4.4 Recommendations for Future Research

It is worth noting that the template for a view is just for a single view. The project documentation should be a *collection* of views and other necessary information. Unfortunately, we have not yet come so far that I could present such a documentation package. This is something we need to study and try out in the future. Nevertheless, I have found a few outlines to this need, and one of the most interesting is made by Ian Gorton (2011, chapter 8.6). The author states that the comprehensive structure for architecture documentation template is as follows:

1. Project Context
2. Architecture Requirements
    a. Overview of Key Objectives
    b. Architecture Use Cases
    c. Stakeholder Requirements
    d. Constraints
    e. Non-functional Requirements
3. Solution
    a. Relevant Architectural Styles
    b. Architecture Overview
    c. Structural Views
    d. Behavioral Views
    e. Implementation Issues
4. Architecture Analysis
    a. Scenario analysis
    b. Risks

In my view, the structure described above is very much in line with the best practices revealed during the thesis process and is forth further study. Architecture use cases help to identify the stakeholders and their requirements. It also makes sense to document the selected architecture styles used for documenting the solution itself. The backbone of the solution documentation is a set of structural views such as module views, C&C views and allocation views. These in turn are complemented by behavioral views like sequence diagrams. The last section is interesting too. To my understanding, architecture analysis is used as a risk mitigation technique. If this is done in a process-like manner, then potential risks can already be identified from the documentation at low cost – before the system is even build or tested. This would be a good topic for further research.

Another interesting topic for further studies is domain-driven design (DDD). I made only one experiment in this approach during the thesis process; however, I already learned plenty about it. DDD made me understand that architecture design can be a valuable method in business analysis and market research. It can help identify where real business problems are and where scarce resources are worth investing. This would be a good further research topic for architects interested in strategic planning.

4.5 Reflections on Research Method

"A poorly designed diary study can involve considerable effort but may yield little useful information." (Bolger, Davis & Rafaeli 2003, 581.)

”Novice student autoethnographers also face considerable difficulties with the research, thesis production, examination and supervision process.” (Doloriert & Sambrook 2012, 88.)

These warnings should be taken seriously but they need not be feared. The thesis writing process has been demanding but also rewarding. When I started doing the research diary my plan was to write a page per day for ten weeks. By the middle of the reporting period, I noticed that there was already plenty of research material. The ease of writing was clearly one of the best aspects of the method. This was probably due to the fact that the threshold for writing diary entries was rather low. Here are a few reasons why. Firstly, I decided to be honest about the problems I encountered at work and incompetence which I felt at times. I wanted to write about *trying* to solve the problem; not just about the solution. Secondly, I did not have to distance myself from the task at hand when writing the research. In other words, there was no need to hide “I” from the entries which was liberating indeed. Often, I was able to concentrate for a long time and got “absorbed” in the studying, diagramming and writing. At its best, the material was produced nearly by itself.

Later, however, I noticed that this was a two-edged sword. Weekly analyses revealed that the plans did not always hold, and topics of diary entries varied from day to day. The third chapter of the research is fragmented to some extent. This emphasizes the importance of a good discussion section in a diary study. I also remember wondering why someone would be interested in my diary entries. In these situations, two things kept me on track. Firstly, the “triggers” that instigated self-reporting were reasonable tight. I did not allow myself to write on *any* subject; each entry had to improve my knowledge of the research topic. A clear research problem and well-defined research questions helped greatly. On the other hand, an evolving idea did not have to be ready to be part of a diary entry. I let the idea develop at its own pace for several days. In retrospect, this fit well with the research method. Chapters (3.3.4-3.4.6) on domain-driven design are a good example of this. Secondly, I constantly reflected what I learned to prior literature. There is no separate literature review in the study because the literature is examined in almost every diary entry. This proved to be a good solution as it accelerated and guided my learning from week to week.

References

Bachmann, F., Bass, L., Clements, P., Garlan, D., Ivers, J., Little, R., Nord, R., & Stafford, J. 2002. *Documenting Software Architecture: Documenting Behavior.* Accessed on 29 March 2020. Retrieved from: https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=5847

Bolger, N., Davis, A., & Rafaeli, E. 2003. Diary Methods: Capturing Life as it is Lived. *Annual Review of Psychology*, 579-616. Accessed on 12 April 2020. Retrieved from: https://www.researchgate.net/publication/10974933_Diary_Methods_Capturing_Life_as_it_is_Lived

Chang, H. 2016. Autoethnography in Health Research: Growing Pains? *Qualitative Health Research*, 26, 443-451. The abstract was accessed on 13 April 2020. Retrieved from: https://www.researchgate.net/publication/294731618_Autoethnography_in_Health_Research_Growing_Pains

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., & Stafford, J. 2010. *Documenting Software Architectures: Views and Beyond.* 2nd ed. Addison-Wesley.

Component Diagram Tutorial. Accessed on 25 March 2020. Retrieved from https://www.lucidchart.com/pages/uml-component-diagram

Data Flow Diagram Symbols. Accessed on 24 February 2020. Retrieved from https://www.lucidchart.com/pages/data-flow-diagram/data-flow-diagram-symbols

Doloriert, C., & Sambrook., S. 2012. Organisational autoethnography. *Journal of Organizational Ethnography*, 83-95. Accessed on 13 April 2020. Retrieved from: https://www.researchgate.net/publication/235305026_Organisational_autoethnography

Ellis, C., Adams, T., & Bochner, A. 2011. Autoethnography: An Overview. *Forum: Qualitative Research*, 12. Accessed on 13 April 2020. Retrieved from: https://www.researchgate.net/publication/48666999_Autoethnography_An_Overview

Flowchart Symbols and Notation. Accessed on 26 February 2020. Retrieved from https://www.lucidchart.com/pages/flowchart-symbols-meaning-explained

Fowler, Martin. 2014. *BoundedContext*. Accessed on 9 March 2020. Retrieved from: https://martinfowler.com/bliki/BoundedContext.html

Gorton, Ian. 2011. *Essential Software Architecture*. 2nd ed. Springer. Retrieved from: https://www.books24x7.com/

Hilliard, R. 2012. *Architecture Viewpoint Template for ISO/IEC/IEEE 42010*. Accessed on 3 March 2020. Retrieved from http://www.iso-architecture.org/42010/templates/42010-vp-template.pdf

Iterations. Accessed on 28 February 2020. Retrieved from https://www.scaledagileframework.com/iterations/

Kananen, J. 2015. *Online research for preparing your thesis: a guide for conducting qualitative and quantitative research online*. JAMK University of Applied Sciences. Accessed on 15 April 2020. Retrieved from https://janet.finna.fi/, Booky.fi

Kaufman, S. 2019. *Guide to Flowchart Symbols, from Basic to Advanced*. Accessed on 26 February 2020. Retrieved from https://www.gliffy.com/blog/how-to-flowchart-basic-symbols-part-1-of-3

Maier, M. & Rechtin, E. *The Art of Systems Architecting*. 2009. 3rd ed. CRC Press.

Malveau, R. & Mowbray, T. 2004. *Software Architect Bootcamp.* 2nd ed. Prentice Hall.

Nonfunctional Requirements. Accessed on 6 March 2020. Retrieved from https://www.scaledagileframework.com/nonfunctional-requirements/

Organizational Memory and Knowledge Repositories. 2010. Accessed on 28 February 2020. Retrieved from https://www.knowledge-management-tools.net/organizational-memory-and-knowledge.php

Sheble, L. & Wildemuth, B. 2009. Research diaries. *Applications of social research methods to questions in information and library science*, 211-221. Accessed on 12 April 2020. Retrieved from: https://www.researchgate.net/publication/279537887_Research_Diaries

Senge, P. 2006. *The Fifth Discipline: The Art & Practice of The Learning Organization.* Doubleday.

Systems and software engineering — Architecture description. 2011a. Accessed on 2 March 2020. Retrieved from https://www.iso.org/standard/50508.html

Systems and software engineering — Architecture description. 2011b. Accessed on 2 March 2020. Retrieved from https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:42010:ed-1:v1:en

Theme templates. Accessed on 19 March 2020. Retrieved from https://shopify.dev/tutorials/develop-theme-templates

Tune, Nick. 2017. *Becoming an Agile Tech Strategist*. Accessed 15 March 2020. Retrieved from https://medium.com/nick-tune-tech-strategy-blog/becoming-an-agile-tech-strategist-5ccd697e7bef

UML Sequence Diagram Tutorial. Accessed on 27 March 2020. Retrieved from https://www.lucidchart.com/pages/uml-sequence-diagram

Vernon, V. *Implementing Domain-Driven Design.* 2013. 1st ed. Addison-Wesley Professional.

What is Domain-Driven Design? 2007. Accessed on 5 March 2020. Retrieved from https://dddcommunity.org/learning-ddd/what_is_ddd/

Zachman, J. *The Framework for Enterprise Architecture: Background, Description and Utility*. The original article was written in 1996. Accessed on 22 Februay 2020. Retrieved from https://www.zachman.com/resources/ea-articles-reference/327-the-

framework-for-enterprise-architecture-background-description-and-utility-by-john-a-zachman